SOFTWARE ENGINEERING WITH COMPUTATIONAL INTELLIGENCE

edited by

Taghi M. Khoshgoftaar

Springer Science+Business Media, LLC

SOFTWARE ENGINEERING WITH COMPUTATIONAL INTELLIGENCE

THE KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE

SOFTWARE ENGINEERING WITH COMPUTATIONAL INTELLIGENCE

edited by

Taghi M. Khoshgoftaar
Florida Atlantic University, U.S.A.

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

Library of Congress Cataloging-in-Publication

Title: **Software Engineering with Computational Intelligence**
Editor: **Taghi M. Khoshgoftaar**

ISBN 978-1-4613-5072-9 ISBN 978-1-4615-0429-0 (eBook)
DOI 10.1007/978-1-4615-0429-0

Table of Contents

Preface vii

Acknowledgment xi

1. **Applying Machine Learners to GUI Specifications in Formulating Early Life Cycle Project Estimations** 1
   Gary D. Boetticher

2. **Applying Fuzzy Logic Modeling to Software Project Management** 17
   Stephen G. MacDonell and Andrew R. Gray

3. **Integrating Genetic Algorithms With Systems Dynamics To Optimize Quality Assurance Effort Allocation** 44
   Balasubramaniam Ramesh and Tarek K. Abdel-Hamid

4. **Improved Fault-Prone Detection Analysis of Software Modules Using an Evolutionary Neural Network Approach** 69
   Robert Hochman, Taghi M. Khoshgoftaar, Edward B. Allen and John P. Hudepohl

5. **A Fuzzy Model and the *AdeQuaS* Fuzzy Tool: a theoretical and a practical view of the Software Quality Evaluation** 101
   Kelly R. Oliveira and Arnaldo D. Belchior

6. **Software Quality Prediction Using Bayesian Networks** 136
   Martin Neil, Paul Krause and Norman Fenton

7. **Towards the Verification and Validation of Online Learning Adaptive Systems** 173
   Ali Mili, Bojan Cukic, Yan Liu and Rahma Ben Ayed

8. **Experimenting with Genetic Algorithms to Devise Optimal Integration Test Orders** 204
   Lionel C. Briand, Jie Feng and Yvan Labiche

9. **Automated Test Reduction Using an Info-Fuzzy Network** 235
Mark Last and Abraham Kandel . . . .

10. **A Genetic Algorithm Approach to Focused Software Usage Testing** 259
Robert M. Patton, Annie S. Wu, and Gwendolyn H. Walton

11. **An Expert System for Suggesting Design Patterns - A Methodology and a Prototype** 287
David C. Kung, Hitesh Bhambhani, Riken Shah and Gaurav Pancholi . . . .

12. **Condensing Uncertainty via Incremental Treatment Learning** 319
Tim Menzies, Eliza Chiang, Martin Feather, Ying Hu, James D. Kiper . . . .

Preface

The constantly evolving technological infrastructure of the modern world presents a great challenge of developing software systems with increasing size, complexity, and functionality. The software engineering field has seen changes and innovations to meet these and other continuously growing challenges by developing and implementing useful software engineering methodologies. Among the more recent advances are those made in the context of software portability, formal verification techniques, software measurement, and software reuse. However, despite the introduction of some important and useful paradigms in the software engineering discipline, their technological transfer on a larger scale has been extremely gradual and limited. For example, many software development organizations may not have a well-defined software assurance team, which can be considered as a key ingredient in the development of a high-quality and dependable software product.

Recently, the software engineering field has observed an increased integration or fusion with the computational intelligence (CI) field, which is comprised of primarily the mature technologies of fuzzy logic, neural networks, genetic algorithms, genetic programming, and rough sets. Hybrid systems that combine two or more of these individual technologies are also categorized under the CI umbrella. Software engineering is unlike the other well-founded engineering disciplines, primarily due to its human component (designers, developers, testers, etc.) factor. The highly non-mechanical and intuitive nature of the human factor characterizes many of the problems associated with software engineering, including those observed in development effort estimation, software quality and reliability prediction, software design, and software testing.

The computational intelligence area provides a software development team with a unique conceptual and algorithmic opportunity of incorporating technologies such as fuzzy logic, neural networks, and evolutionary computation to address open software engineering problems. The basic purpose of incorporating CI technologies into the various phases of software development and analysis is to address the problems arising due to imprecise measurement and uncertainty of information.

This book presents a collection of twelve articles selected from peer-reviewed papers that focus on computational intelligence in software engineering. CI technologies have been applied to solve software engineering issues arising from the ever-increasing complexity of software systems descriptions and analysis. These papers were initially selected for a special volume of the Annals of Software Engineering (ASE) journal, published by Kluwer Academic Publishers. Titled "Computational Intelligence in Software Engineering", this special volume was destined to be Volume 15 of the ASE. However, due to unfortunate circumstances, the ASE journal was discontinued after the publication of Volume 14. In light of the considerable efforts put in by the various reviewers, volunteers, and myself toward the realization of the special volume, Kluwer decided on publishing the same in the form of this book. I am thankful to them for recognizing our efforts. Twelve high-quality research papers were chosen based on a rigorous peer-review and selection process, from over seventy submissions received for the special volume of ASE. Most submissions were reviewed by at least four reviewers familiar with the respective subject matter. Osman Balci, the Editor-in-Chief of ASE, handled the review process for the included paper by Hochman, Khoshgoftaar, Allen, and Hudepohl.

The contributions of this book are generally grouped into four software engineering categories: software project management and effort estimation; software quality assurance and estimation; software testing, verification, and validation; and, software design. All the papers focus on applying computational intelligence techniques to address software engineering problems.

The group of articles related to software project management and effort estimation consists of two papers. The paper by Boetticher presents a process of applying machine learners, in particular neural networks, to formulate effort estimation models early in the software life cycle. The approach presented is alternative to the traditional software cost estimation models, COCOMO II and Function Points Analysis. Empirical case studies are based on input and output measures extracted from GUI interface specification documents. The goal of this paper is to define a tool that deterministically constructs accurate early life cycle estimation models. The paper by MacDonell and Gray presents a study of applying fuzzy logic to the software project management problem. A software toolset is constructed that enables data, classes and rules to be defined for estimation problems such as project risk, product size, and project effort based on a variety of input parameters. The effectiveness of the fuzzy logic-based toolset is demonstrated by applying it to two data sets, the first related to software sizing and the second to effort estimation.

The next group of papers (four) is related to software quality assurance and software quality estimation. The first among these, by Balasubramaniam and Abdel-Hamid, focuses on improving the decision support system with a genetic algorithm to optimize the decision variable. A decision support system based on the system dynamics simulation model has been developed to support decision-making regarding the optimal allocation of effort toward software quality assurance. The overall objective of this study is to present a tool to support the software project manager in efficiently allocating the quality assurance effort throughout the software development lifecycle. A case study is based on a real project that was conducted at one of NASA's space flight centers.

The paper by Hochman, Khoshgoftaar, Allen, and Hudepohl describes an application of genetic algorithms to the evolution of optimal or near-optimal backpropagation neural networks for the software quality classification problem. Predicting the quality of modules as either fault-prone or not fault-prone can support the decision making process regarding the allocation of software quality improvement resources. The ultimate goal in this informed allocation of software project resources is to contain costs and maintain schedules with minimal impact on software quality. It is suggested that evolutionary neural networks can be used to successfully attack a broad range of data-intensive software engineering problems, where traditional methods have been used almost exclusively.

The paper by Olivera and Belchior introduces a fuzzy logic-based model for software quality evaluation and its implementation, the *AdaQuaS* Fuzzy tool. The study clearly demonstrates how fuzzy logic is more suitable for software engineering decision making processes in which inherent subjectivity and inconsistencies are found due to the human component. Two empirical case studies are presented: the first is the software quality evaluation process of e-commerce website; the second is an evaluation of software requirements specification quality. The paper by Neil, Krause, and Fenton is based on the aim of producing a single model that combines the diverse forms of, often causal, evidence available in software development in a more natural and efficient way than done in previous related studies. They use Bayesian Belief Networks as the appropriate formalism for demonstrating the same. The authors argue that the causal or influence structure of these models more naturally mirrors the real world sequence of events that can be achieved with other formalisms.

The third group of papers, related to software testing, verification, and validation, consists of four papers. The one by Mili, Cukic, Liu, and Ayed presents a framework for reasoning about on-line learning systems, which they envision as a candidate technology for their verification and validation. Their discussion is based on a practical problem associated with adaptive systems, i.e., due to the constantly evolving nature of adaptive systems, they are inherently more difficult to verify/validate. The paper by Briand, Feng, and Labiche presents an improved strategy to devise optimal integration test orders in object-oriented systems in the presence of dependency cycles. The goal of their study is to minimize the complexity of stubbing during integration testing, which has been shown to be a major source of software testing costs. The strategy to achieve the goal is based on the combined use of inter-class coupling measurement and genetic algorithms, which are used to minimize cost functions based on coupling measurement.

The paper by Last and Kandel presents an attempt to automate a common task in black-box software testing, namely reducing the number of combinatorial test cases. Their proposed approach is based on automated identification of relationships between inputs and outputs of a data-driven application. The input variables relevant to each output are extracted by the proposed data mining algorithm, called the info-fuzzy network. The case study is that of a typical business software application program. The last paper in this group, by Patton, Wu, and Walton, besides providing a basic introduction to genetic algorithms, presents a genetic algorithm-based approach to focused software usage testing. A genetic algorithm is used to select additional test cases to focus on the behavior around the initial set of test cases in order to assist in identifying and characterizing the types of test cases that do/do not induce system failures. It is shown that the approach presented supports increased test automation and provides increased evidence to support reasoning about the overall quality of the software.

The next paper in this book is related to computational intelligence as applied to software design problems. The authors, Kung, Bhambani, Shah, and Pancholi present a methodology for constructing expert systems which can suggest software design patterns to solve design problems as stated by the software design team. More specifically, the expert system selects a design pattern through dialog with the software designer to narrow down the possible choices. Moreover, classification and heuristics have been used to improve the selection effectiveness. The final paper, by Menzies, Chiang, Feather, Hu, and Kiper, presents a machine-learning approach of condensing uncertainty of various prediction problems. Their method is based on incremental treatment learning and their previously proposed funnel theory.

Taghi M. Khoshgoftaar

Florida Atlantic University
Boca Raton, Florida, USA

Acknowledgment

I would like to take this opportunity to thank all the reviewers for their patience and effort in the reviewing process. I thank Susan Lagerstorm-Fife for her invaluable assistance in publishing these papers in the form of a book, despite the termination of the Annals of Software Engineering journal. I would like to express my appreciation to Sharon Pallechi at Kluwer Academic Publishers for her courteous cooperation in the production of this book. Special thanks to my graduate students: Pierre Rebours for his help with type setting and compilation of the papers; Eric Geleyn and Laurent Nguyen for their assistance with correspondence during the review process, and Naeem Seliya for his editorial reviews and assistance in producing this book. Last but not the least, I am deeply grateful to Osman Balci, the Editor-in-Chief of the ASE, for inviting me to edit this special volume of the ASE journal. Taghi M. Khoshgoftaar was supported in part by the National Science Foundation grant CCR-9970893.

Taghi M. Khoshgoftaar

Florida Atlantic University
Boca Raton, Florida, USA
December 2002

Applying Machine Learners to GUI Specifications in Formulating Early Life Cycle Project Estimations

Gary D. Boetticher

Department of Software Engineering
University of Houston, Clear Lake
2700 Bay Area Boulevard, Houston, TX 77058, USA
+1 281 283 3805
boetticher@cl.uh.edu

ABSTRACT

Producing accurate and reliable early life cycle project estimates remains an open issue in the software engineering discipline. One reason for the difficulty is the perceived lack of detailed information early in the software life cycle. Most early life cycle estimation models (e.g. COCOMO II, Function Point Analysis) use either the requirements document or a size estimate as the foundation in formulating polynomial equation models. This paper explores an alternative approach using machine learners, in particular neural networks, for creating a predictive effort estimation model. GUI-specifications captured early in the software life cycle serve as the basis for constructing these machine learners. This paper conducts a set of machine learning experiments with software cost estimation empirical data gathered from a "real world" eCommerce organization. The alternative approach is assessed at the program unit level, project subsystem level, and project level. Project level models produce 83 percent average accuracy, pred(25), for the client-side subsystems.

KEYWORDS

Machine learning, machine learners, requirements engineering, software engineering, neural networks, backpropagation, software metrics, effort estimation, SLOC, project estimation, programming effort.

1. INTRODUCTION

One of the most important issues in software engineering is the ability to accurately estimate software projects early in the life cycle. Low estimates result in cost overruns. High estimates equate to missed financial opportunities.

From a financial context, more than $300 billion is spent each year on approximately 250,000 software projects [30]. This equates to an average budget of $1.2 million. Coupling these facts with Boehm's observation [2] that project estimates range from 25 to 400 percent early in the life cycle indicates a financial variance of $300 thousand to $4.8 million.

Why does such a high variance exist? One primary reason is the severe lack of data early in the life cycle. In the embryonic stage of a software project the only available artifact is a requirements document. This high-level document provides relatively few metrics (e.g., nouns, verbs, adjectives, or adverbs) for estimating a project's effort. Due to the complexity and ambiguity of the English language, formulating an accurate and reliable prediction, based upon a requirements document, is a nearly impossible task.

Despite this high variance, the ability to generate accurate and reliable estimates early in software life cycle is extremely desirable. Many IT managers are under pressure to offer relatively narrow ranges of estimates regarding anticipated completion rates.

The software engineering discipline recognizes the importance of building early life cycle estimation models. The traditional approach involves formulating a polynomial equation based upon empirical data. Well-known equations include the COnstructive COst MOdel II (COCOMO II) and Function Point Analysis. Each has produced reasonable results since their inception [13, 28]. However, there are several drawbacks in using these well-known equations.

Using these equations is a time-consuming process. The complexity of each requires extensive human intervention and is subject to multiple interpretations [21, 31]. What is needed is an alternative approach for generating accurate estimates early in the software life cycle. An approach which produces accurate estimates, is automated to avoid subjective interpretation; and is relatively simple to implement.

This paper describes a process of applying machine learners, in particular neural networks, in formulating estimation models early in the software life cycle. A series of empirical experiments are based on input and output measures extracted from four different 'real world' project subsystems. The input measures for each experiment are derived by utilizing the GUI interface specification document. The GUI interface document offers the advantage of being an early life cycle artifact rich in objective measures for building effort estimation models.

The set of experiments use 109 different data samples (or program units). Each program unit corresponds to a form consisting of up to twelve different types of widgets (e.g., edit boxes, buttons). Extracted widget counts serve as the input measures. The output measure is the actual, not estimated, effort expended in developing that particular program unit.

Section 2 provides background information and motivation for using machine learners in project estimation. Section 3 discusses related research in the area of machine learning applied to effort estimation. Section 4 describes a set of machine learning experiments. Section 5 offers a discussion of the experiments. And section 6 draws several conclusions and describes future directions.

2. BACKGROUND

Different techniques for cost estimation have been discussed in the literature [2, 16, and 18]. Popular approaches include: algorithmic and parametric models, expert judgment, formal and informal reasoning by analogy, price-to-win, top-down, bottom-up, rules of thumb, and available capacity.

Two well-known parametric approaches for early life cycle estimation are COCOMO II and Function Point Analysis (FPA).

The COCOMO II equation embeds many project parameters within the equation. It is defined as follows [10]:

$$\text{Effort} = A * (\text{Size})^{B} * \text{EM} \tag{1}$$

where

Effort refers to the Person Months needed to complete a project

A represents the type of project. There are three possible values for this parameter.

Size is defined by using a SLOC estimate or Function Point Count.

B is a derived metric which includes the sum of five cost driver metrics.

EM is an abbreviation for Effort Multiplier. The COCOMO II equation defines seven effort multipliers for early life cycle estimating.

A difficulty in applying the COCOMO II equation is managing the very large solution space. In the early life cycle version, there are 3 options for project type, 5^5 options for the cost drivers, and 5^7 options for the effort multipliers. Multiplying all the options together reveals a search space of 732,421,875 different settings. This excludes the effort value supplied for the *Size* parameter.

A more fundamental problem with COCOMO II is that it requires an estimate for the *size* of the project represented by SLOC of Function Points.

If the size of a project were actually known early in the life cycle, then it would be easy to formulate a reasonable effort estimate.

Another parametric approach is Function Point Analysis (FPA). This process starts with the requirements document where a user identifies all processes. Each process is categorized into one of five function types; different Record Element Type, Data Element Types; and File Types Referenced. Based upon the settings chosen, the equation produces an *Unadjusted Function Point* (UFP) for each process. There are seven possible values for each UFP ranging from 3 through 15.

The next step involves defining the *Global System Characteristics* (GSC). Collectively, there are 14 different GSC parameters with 5 possible settings for a total search space of 6,103,515,625 options. The total GSC may range from 0 through 70.

After applying several mathematical operations to the UFP, it is multiplied by the total GSC to produce the final Adjusted Function Point. A software project with only one function point may range from 1.95 to 20.25 Adjusted Function Points. Assuming a mean of 11.1, this produces a variance of 83.7. This Adjusted Function Point variance does not compare well with Boehm's early life cycle variance of 4 [2].

Assuming a perfect Adjusted Function Point value is determined, the next step in the FPA requires the model builder (presumably a project domain expert) to define a constant by which to multiply the final Adjusted Function Point total. This last step, which is totally subjective, is the most critical step in the process and very sensitive to distortion.

The complex nature of COCOMO II and FPA suggests the need for an alternative approach to early life cycle project estimation.

One approach would be to formulate an early life cycle model using a machine learning algorithm. There are different types of machine learners including predictors, classifiers, and controllers. Since this is a predictor type problem, a neural network approach is chosen for conducting a series of experiments.

The goal of these experiments is to define a tool that deterministically constructs an accurate early life cycle estimation model. Deterministically means that there are no subjective measures introduced into the modeling process.

To establish a context for the application of machine learners to software project estimation; the following section describes previous research in this area.

3. RELATED RESEARCH

Related research consists of the utilization of various types of machine learners for predicting project effort. Also, there had been some previous research in using GUI metrics for estimating project effort. This section describes both of these contexts in terms of machine learning algorithms deployed, if applicable, and results achieved.

In [1, 15, 20, 24, and 25], a Case-Based Reasoning (CBR) approach is adopted in constructing a cost model for the latter stages of the development life cycle. Delany [12] also uses a CBR approach applied early in development life cycle.

Chulani [9] uses a Bayesian approach to cost modeling and generates impressive results. He collects information on 161 projects from commercial, aerospace, government, and non-profit organizations [9]. The COCOMO data sets contain attributes that, for the most part, can be collected early in the software life cycle (exception: COCOMO requires source lines of code which must be estimated). Regression analysis was applied to the COCOMO data set to generate estimators for software project effort. However, some of the results of that analysis were counter-intuitive. In particular, the results of the regression analysis disagreed with certain domain experts regarding the effect of software reuse on overall cost.

To fix this problem, a Bayesian learner was applied to the COCOMO data set. In Bayesian learning, a directed graph (the belief network) contains the probabilities that some factor will lead to another factor. The probabilities on the edges can be seeded from (e.g.) domain expertise. The learner then tunes these probabilities according to the available data. Combining expert knowledge and data from the 161 projects yielded an estimator that was within 30% of the actual values, 69% of the time [9]. It is believed that the above COCOMO result of pred(30) = 69% is a high-watermark in early life cycle software cost estimation.

Cordero [11] applies a Genetic Algorithm (GA) approach in the tuning of COCOMO II.

Briand [8] introduces optimized set reduction (OSR) in the construction of software cost estimation model.

Srinivasan [29] builds a variety of models including neural networks, regression trees, COCOMO, and SLIM. The training set consists of COCOMO data (63 projects from different applications). The training models are tested against the Kemerer COCOMO data (15 projects, mainly business applications). The regression trees outperformed the COCOMO and the SLIM model. The neural networks and function point-based prediction models outperformed regression trees.

Samson [26] applies neural network models to predict effort from software sizing using COCOMO-81 data. The neural network models produced better results than the COCOMO-81.

Wittig *et al.* [32] estimated development effort using a neural network model. They achieved impressive results of 75 percent accuracy pred(25).

Boetticher [6] conducted more than 33,000 different neural network experiments on empirical data collected from separate corporate domains. The experiments assessed the contribution of different metrics to programming effort. This research produced a cross-validation rate of 73.26%, using pred(30).

Hodgkinson [19] adopted a top-down approach using a neurofuzzy cost estimator in predicting project effort. Results were comparable to other techniques including least-squares multiple linear regression, estimation via analogy, and neural networks.

Lo *et al.* [23] constructed a GUI effort estimation multivariate regression model using 33 samples. Independent variables consisted of GUI metrics classified into 5 groups: static widgets (labels), data widgets not involving lists (edit boxes, check boxes, radio buttons), data widgets involving lists (list boxes, memo boxes, file lists, grids, combo boxes), action widgets involving the database (buttons), and action widgets not involving the database (buttons). Instead of using the actual effort values for the dependent variable, estimates from 4 experts with a least 1 year of experience were averaged. The average of these estimates ranged from 3 to 48 hours. Variance of the expert's estimates is not presented in the paper. The initial internal results were pred(25) = 75.7% and MARE 20.1%. External results (against another system) yielded were pred(25) = 33.3% and MARE = 192%.

4. MACHINE LEARNING EXPERIMENTS

4.1. General Description

This section describes a series of machine learning experiments based on data gathered from four 'real-world' project subsystems.

Prior to conducting the experiments, it was necessary to decide which ML approach to adopt. A neural network paradigm for creating models seemed like a natural choice. This decision was based upon the author's previous successes using neural networks to model software metrics [3, 4, 5, 6, and 7].

Advantages of using neural networks include [17]: the ability to deal with domain complexity, ability to generalize, along with adaptability, flexibility, and parallelism.

There is also support in the literature for applying neural networks in estimation tasks [22, 26, and 29]. However, some researchers consider the relative merits of neural nets over other machine learning techniques (e.g. decision tree learning) an open issue [27].

4.2. Neural Network Overview

A supervised neural network can be viewed as a directed graph composed of nodes and connections (weights) between nodes. A set of vectors, referred to as a training set, is presented to the neural network one vector at a time. Each vector consists of input values and output values. In Figure 1, the inputs are $\boldsymbol{x}_0$ through $\boldsymbol{x}_{N-1}$ and the output is $\boldsymbol{y}$. The goal of a neural network is to characterize a relationship between the inputs and outputs for the whole set of vectors. During the training of a neural network, inputs from a training vector propagate throughout the network. As inputs traverse the network, they are multiplied by appropriate weights and the products are summed up. In Figure 1, this is $\boldsymbol{w}_i \cdot \boldsymbol{x}_i$. If the summation exceeds some specified threshold for a node, then output from that node serves as input to another node. This process repeats until the neural network generates an output value for the corresponding input portion of a vector. This calculated output value is compared to the desired output and an error value is determined. Depending on the neural network algorithm, either the weights are recalibrated after every vector, or after one pass (called an epoch) through all the training vectors. In either case the goal is to minimize the error total. Processing continues until a zero error value is achieved, or training ceases to converge. After training is properly completed, the neural network model which characterizes the relationship between inputs and outputs for all the vectors is embedded *within the architecture (the nodes and connections) of the neural network.* After successful completion of training, a neural network architecture is frozen and tested against an independent set of vectors called the test set. If properly trained, the neural network produces reasonable results against the test suite.

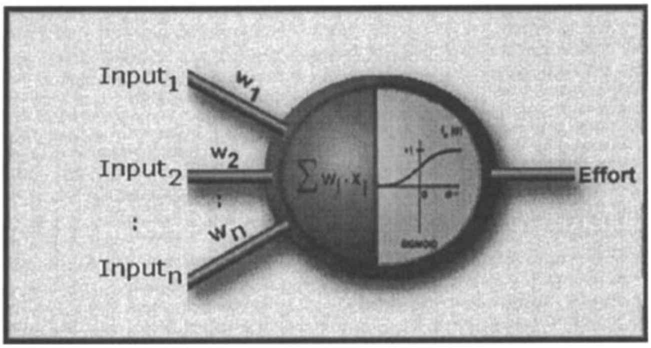


Figure 1. Sample Neural Network.

All experiments utilize a variant of the backpropagation neural network, called the quickprop. The quickprop algorithm converges much faster than a typical backpropagation approach [14]. It uses the higher-order derivatives in order to take advantage of the curvature [14]. The quickprop algorithm uses second order derivatives in a fashion similar to Newton's method. Using quickprop in all the experiments also ensures stability and continuity.

4.3. Description of Experiments

Four datasets were used in the experiments. The GUI metrics were extracted from one of four major subsystems of an electronic commerce (procurement) product used in the process industry. Table 1 shows each major system along with the number of program units.

Major Subsystem	Number of program units
Buyer Administrator	7
Buyer	60
Distribution Server	10
Supplier	32
TOTAL	**109**

Table 1. Description of the major subsystems.

Each program unit consists of a GUI form along with corresponding code written in Delphi.

In the context of neural networks, a program unit is referred to as a vector. Each vector consists of a set of inputs along with a set of outputs. Each vector contained twelve input parameters based upon GUI categories described below. These include: Buttons; Charts; Check boxes/radio buttons; Combo boxes; Grid (string grid, database grid); Grid Tabs; Edit boxes; Labels; Memo/List boxes; Menu bars; Navigation bars; and Trees.

The grid tabs refers to how many tabs were available for each grid. For example, the default is three (worksheets) in Microsoft© EXCEL.

The output consists of the actual, not estimated, effort required for developing each program unit. Effort values ranged from 1 to 160 hours. All program units were developed by a single developer. This reduces the impact of the human element in terms of various skill and knowledge levels in the model formulation process.

All experiments use a fully-connected neural network architecture of 12-5-11-5-1 (see Figure 2), meaning twelve inputs, one layer of five hidden nodes, followed by another hidden layer of eleven hidden nodes, followed by another hidden layer of five nodes, then an output layer of one node.

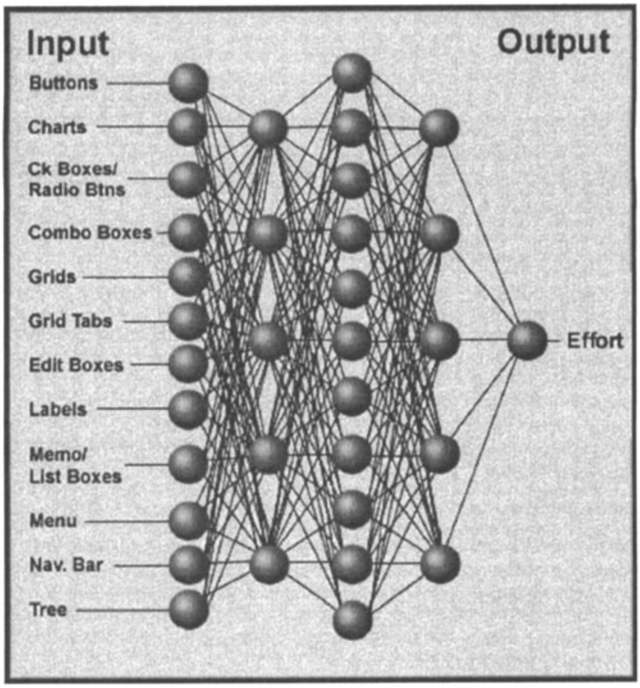


Figure 2. 12-5-11-5-1 Neural Network Architecture.

In order to minimize experimental variance among experiments, we standardized the experimental process. Different components of each neural network model remain constant. Alpha, which represents how quickly a neural network learns, may range from zero to one. Alpha for these experiments is always one. Momentum, a variable which helps neural networks break out of local minima, may also range from zero to one. Momentum is also always set to 1. The threshold function is a function associated with each node after the input layer. Function selection determines when a node fires. Firing a node essentially propagates a value further through the network. All experiments use an asymmetrical sigmoid function as a threshold function.

One scan through the training data is considered an epoch. Each experiment iterates through 10,000 epochs. Initial trials indicated that 5,000 to 7,000 epochs were sufficient for determining the highest correlation along with the highest accuracy (with respect to the test data). The "most accurate test results" is defined as number of correct matches within 25 percent, or pred(25), of the actual effort values for the test vectors.

4.4. Experiments and Results

Four different neural network experiments are performed for each subsystem. For each set of experiments, data from one of the subsystems is treated as a test suite and the data from the other three subsystems is combined into a training set. Each experiment is performed ten times in order to discount any outliers. After each experiment all the weights in the neural network are reset.

Table 2 presents the Pred(25) and MARE for each of the four types of experiments. The Pred(25) of 64.3 percent for the 'Buyer Administrator' test set is reasonable, however the other Pred(25) values are low relative to [9, 23]. The relatively low values for the Pred(25) may be attributed to range of effort values, 1 to 160 hours. Nineteen percent of all the vectors had an effort of one. As a consequence, the neural network experienced difficulty in adequately approaching these low effort values. Secondly, the test sets were organized by subsystem. This led to extrapolation issues. A total of 15 vectors from three different test sets contained maximum values for one or more inputs/output. The only way to avoid any extrapolation problems would be to continuously retain the 15 vectors in the training set. This is not a realistic solution. Finally, for some of the input metrics, there were less than three instances of values. Essentially, the corresponding metric contributed little to the training process.

The relatively large range of effort values, along with the large percentage of effort values less than or equal to five (65 percent) inflated the MARE values.

	Buyer Admin		Buyer Client		Distribution Server		Supplier Client	
RUN	Pred (25)	MARE	Pred (25)	MARE	Pred (25)	MARE	Pred (25)	MARE
1	71%	54%	32%	231%	40%	212%	34%	176%
2	57%	54%	33%	355%	50%	160%	38%	126%
3	57%	98%	32%	258%	50%	319%	44%	179%
4	57%	72%	23%	385%	60%	84%	41%	200%
5	57%	89%	27%	248%	50%	66%	38%	288%
6	71%	38%	33%	210%	50%	176%	41%	171%
7	71%	82%	25%	253%	50%	74%	38%	189%
8	71%	44%	18%	312%	50%	55%	38%	198%
9	71%	53%	23%	254%	50%	492%	38%	178%
10	57%	103%	25%	652%	60%	68%	44%	131%
AVE	**64.3%**	**68.5%**	**27.2%**	**316%**	**51%**	**172%**	**39.1%**	**184%**

Table 2. Results from initial experiments.

One question is whether the results are any different when perceived from the subsystem (project) level. Viewing the results from a subsystem perspective, as opposed to a program unit perspective, dramatically improves upon the results. Table 3 shows the Pred(25) and MARE for each of the four major subsystems. The results are determined by summing the calculated effort values for each program unit for each individual experiment. Hence three of the four models produced estimates 80 percent or higher with MARE values less than 18 percent. The best case generates a Pred(25) of 90 percent and a MARE of 12.2 percent.

Subsystem	Pred(25)	MARE
Buyer Administrator	80%	17.6%
Buyer Client	80%	14.6%
Distribution Server	20%	96.7%
Supplier Client	90%	12.2%

Table 3. Aggregate analysis of the subsystems.

A natural extension of the subsystem results would be to aggregate them into corresponding project results. Table 4 presents two worst-case scenarios and one average-case scenario. For the worst-case scenarios the minimum average efforts and maximum average efforts are totaled. The average-case scenario is the average estimate (for the ten experiments) for each subsystem.

Subsystem	Worst case Min.	Worst Case Max.	Ave. Cases	Actual Effort
Buyer Admin.	158	289	220	215
Buyer Client	958	1660	1313	1202
Dist. Server	114	246	170	307
Supplier Client	505	790	644	576
TOTAL	**1735**	**2985**	**2347**	**2300**

Table 4. Aggregate analysis of the project.

Thus, in the worst-case scenarios, the collective estimates range from 24.6% below the actual project effort to 29.8% above the actual project effort. The average-case estimate is within 2% of the actual project effort.

5. DISCUSSION

The vectors were grouped according to the project subsystems, rather than applying a statistical process for organizing the vectors. This followed the natural contours of the project at the expense of generating artificially better results.

One question that persists is, "Why are the results so low for the distribution server presented in table 3?" In general, server-side applications are not intended to be interactive. As a consequence, GUI-based metrics might not be appropriate for server-side software. However, the server-side metrics did not taint the results presented in table 4.

This work extends the previous research of *Lo et al* [23] by assessing more data in greater detail using better effort values. See Table 5 below.

Category	**Lo [23]**	**Current work**
Data Samples	33	109
Number of GUI metrics utilized	5	12
Does formulating GUI metrics require human interpretation?	Yes	No
Model type	Multivariate regression	Neural networks
Nature of effort values	Defined through expert estimates	Based on actual effort values
Best results	Pred(25) = 75.7%, MARE = 20.1%	Pred(25) = 90%, MARE = 18%

Table 5. Comparison of current with previous research.

Extrapolation issues frequently arose during the conducting of experiments. Adding more data might reduce the frequency of extrapolation, but it will never eliminate the problem. The extrapolation issues did not seem to affect the results produced at the subsystem and overall project levels.

It is worth noting the software environment from which this data emerged. This organization did not have a formal process. Most likely it would be characterized as a one in the context of the Capability Maturity Model. Thus, the experiments show that it is possible to construct very reasonable early life cycle project estimates in light of a poorly defined process.

6. CONCLUSIONS AND FUTURE DIRECTIONS

This research describes a process of formulating early life cycle project estimates based on GUI specifications. Twelve different types of widgets are counted. Most, if not all, of the widgets are frequently used and well understood within the software industry.

Using well-defined widgets simplifies the measurement gathering process. Thus, the learning curve is rather shallow (as compared to COCOMO II or FPA) for understanding the model formulation process.

The model formulation process is repeatable since there is no subjectivity involved in counting the widgets.

The results at the program unit level seemed low. However, the results improved dramatically when viewed from the project subsystem level. Three of the four subsystems produced a Pred(25) of 80% or higher and a MARE of 18% or lower.

One strategy to improve upon the results at the program unit level would be to reduce the number of classes for the effort values. Thus a set of actual effort values ranging from 10 to 15 hours may be collapsed into 12.5 hours.

Since the literature describes various applications of Machine Learner in effort estimation, it would be plausible to conduct additional experiments using other machine learning algorithms.

The process does not fare very well in situations where development is computationally complex with little or no GUI specifications. This is evident in the server-side results. One future activity would be to integrate the neural network-based GUI effort estimation approach with an algorithmic approach, such as COCOMO II or FPA. This could reduce the complexity related to COCOMO II and FPA and extend the neural network GUI approach to accommodate non-GUI software development.

ACKNOWLEDGEMENTS

The author wishes to express his deepest gratitude for the insightful comments and constructive criticism from the peer reviewers.

REFERENCES

[1] Bisio, R., F. Malabocchia, "Cost Estimation of Software Projects through Case-Base Reasoning". *Case-Based Reasoning Research and Development. First International Conference*, ICCBR-95 Proceedings, 1995, Pp.11-22.

[2] Boehm, B., *Software Engineering Economics*, Englewood Cliffs, NJ, Prentice-Hall, 1981.

[3] Boetticher, G., K. Srinivas and D. Eichmann, "A Neural Net-Based Approach to Software Metrics", *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering*, June 1993, Pp. 271-274. Available from http://nas.cl.uh.edu/boetticher/publications.html

[4] Boetticher, G. and D. Eichmann, "A Neural Net Paradigm for Characterizing Reusable Software", *Proceedings of the First Australian Conf. on Software Metrics*, November 1993, Pp. 41-49. Available from http://nas.cl.uh.edu/boetticher/publications.html

[5] Boetticher, G., "Characterizing Object-Oriented Software for Reusability in a Commercial Environment", *Reuse '95 Making Reuse Happen – Factors for Success*, Morgantown, WV, August 1995. This paper is available at: http://nas.cl.uh.edu/boetticher/publications.html

[6] Boetticher, G., "An Assessment of Metric Contribution in the Construction of a Neural Network-Based Effort Estimator", Second Int. Workshop on Soft Computing Applied to Soft. Engineering, 2001. Available at http://nas.cl.uh.edu/boetticher/publications.html

[7] Boetticher, G., "Using Machine Learning to Predict Project Effort: Empirical Case Studies in Data-Starved Domains", Workshop on Model-Based Requirements Engineering, 2001. Available at http://nas.cl.uh.edu/boetticher/publications.html

[8] Briand, Lionel C., Victor R. Basili, and William Thomas. "Pattern Recognition Approach for Software Engineering Data Analysis". *IEEE Trans. on Soft. Eng.*, November 1992, Pp. 93-942.

[9] Chulani, S., and Boehm, B., and B. Steece, "Bayesian Analysis of Empirical Software Engineering Cost Models", *IEEE Transaction on Software Engineering*, 25 4, July/August, 1999.

[10] COCOMO II Model Definition Manual, 1999. Available from the following link: http://sunset.usc.edu/research/COCOMOII/Docs/modelman.pdf

[11] Cordero, R., M. Costramagna, and E. Paschetta. "A Genetic Algorithm Approach for the Calibration of COCOMO-like Models", *12th COCOMO Forum*, 1997.

[12] Delany, S.J., P. Cunningham, "The Application of Case-Based Reasoning to Early Project Cost Estimation and Risk Assessment", *Department of Computer Science, Trinity College Dublin*, TDS-CS-2000-10, 2000.

[13] Devnani-Chulani, Sunita, Clark, B., Barry Boehm, "Calibration Approach and Results of the COCOMO II Post-Architecture Model", ISPA, June 1998.

[14] Fahlman, S.E., *An Empirical Study of Learning Speed in Back-Propagation Networks*, Tech Report CMU-CS-88-162, Carnegie Mellon University, September 1988.

[15] Finnie, G.,R., Wittig, G.,E., J.M. Desharnais, "Estimating software development effort with case-based reasoning", *Proceedings of International Conference on Case-Based Reasoning*, D. Leake, E. Plaza, (Eds), 1997, Pp.13-22.

[16] Heemstra, F. "Software Cost Estimation", *Information and Software Technology*, October 1992, Pp. 627-639.

[17] Hertz, J., Krogh A., R.G. Palmer, *Introduction to the Theory of Neural Computation*, Addison Wesley, New York, 1991.

[18] Hihn, J., H. Habib-Agahi, "Cost Estimation of Software Intensive Projects: A Survey of Current Practices", *Proceedings of the International Conference on Software Engineering*, 1991, Pages 276-287.

[19] Hodgkinson, A.C., Garratt, P.W., "A Neurofuzzy Cost Estimator", *Proc. 3rd International Conf. Software Engineering and Applications (SAE)*, 1999, pp. 401-406.

[20] Kadoda, G., Cartwright, M., Chen, L. and Shepperd, M., "Experiences Using Case-Based Reasoning to Predict Software Project Effort", *Empirical Software Engineering Research Group Technical Report*, Bournemouth University, January 27 2000.

[21] Kemerer, Chris, "Reliability of Function Points Measurement: A Field Experiment", *Communications of the ACM 36*, 2 (February 1993), Pp. 85-97.

[22] Kumar, S., Krishna, B. A., Satsangi, P.J., "Fuzzy Systems and Neural Networks in Software Engineering Project Management", *Journal of Applied Intelligence*, 4, 1994, Pp. 31 - 52.

[23] Lo, R., Webby, R., R. Jeffrey, "Sizing and Estimating the Coding and Unit Testing Effort for GUI Systems", *Proceedings of the 3rd International Software Metrics*, Los Alamitos: IEEE Computer Society Press, 166-173, 1996.

[24] Mukhopadhyay, Tridas, and Sunder Kekre, "Software effort models for early estimation of process control applications", *IEEE Transactions on Software Engineering*, 18 (10 October), 1992, Pp. 915-924.

[25] Prietula, M., S. Vicinanza, T. Mukhopadhyay, "Software effort estimation with a case-based reasoner", *Journal of Experimental and Theoretical Artificial Intelligence*, 8(3-4), 1996, Pp. 341-363.

[26] Samson, B., Ellison, D., Dugard, P., "Software Cost Estimation Using an Albus Perceptron", Information and Software Technology, 1997, pp. 55-60.

[27] Shavlik, J.W., Mooney, R.L., and G.G. Towell, "Symbolic and Neural Learning Algorithms: An Experimental Comparison", *Machine Learning*, 1991, Pp. 111-143.

[28] Siddiqee, M. Waheed. "Function Point Delivery Rates Under Various Environments: Some Actual Results", 259-264. *Proceedings of the Computer Management Group's International Conference*. San Diego, CA, December 5-10, 1993.

[29] Srinivasan, K., and D. Fisher, "Machine Learning Approaches to Estimating Software Development Effort", *IEEE Trans. Software Engineering*, February, 1995, Pp. 126-137.

[30] The Standish Group, Chaos Chronicles III, The Standish Group, January, 2003.

[31] Wittig, G. E., G.R. Finnie, "Software Design for the Automation of Unadjusted Function Point Counting", *Business Process Re-Engineering Information Systems Opportunities and Challenges, IFIP TC8 Open Conference.* Gold Coast, Queensland, Australia, May 8-11, 1994, Pp. 613-623.

[32] Wittig, G., G. Finnie, "Estimating software development effort with connectionist models", Information and Software Technology, 1997, pp. 469-476.

Applying Fuzzy Logic Modeling to Software Project Management

Stephen G. MacDonell[1] and Andrew R. Gray[2]

[1]School of Information Technology
Auckland University of Technology
Private Bag 92006, Auckland, New Zealand
stephen.macdonell@aut.ac.nz

[2]Department of Information Science
University of Otago
P.O. Box 56, Dunedin, New Zealand

ABSTRACT

In this paper we provide evidence to support the use of fuzzy sets, fuzzy rules and fuzzy inference in modeling predictive relationships of relevance to software project management. In order to make such an approach accessible to managers we have constructed a software toolset that enables data, classes and rules to be defined for any such relationship (e.g. determination of project risk, or estimation of product size, based on a variety of input parameters). We demonstrate the effectiveness of the approach by applying our fuzzy logic modeling toolset to two previously published data sets. It is shown that the toolset does indeed facilitate the creation and refinement of classes of data and rules mapping input values or classes to outputs. This in itself represents a positive outcome, in that the approach is shown to be capable of incorporating data and knowledge in a single model. The predictive results achieved from this approach are then compared to those produced using linear regression. While this is not the principal aim of the work, it is shown that the results are at least comparable in terms of accuracy, and in specific cases fuzzy logic modeling outperforms regression. Given its other appealing characteristics (for instance, transparency, robustness, incorporation of uncertainty), we believe that fuzzy logic modeling will be useful in assisting software personnel to further improve their management of projects.

KEYWORDS

Fuzzy logic modeling, project management, software metrics.

1. INTRODUCTION

Effective management of projects remains a significant challenge in software engineering practice. While technical and marketing issues may have a strong influence over a project's success, poorly managed projects are more likely to fail than succeed, no matter how clever the technology or how well tailored the product to the market's needs. In this regard the use of effective management models in classifying and predicting important project management parameters (for example, defect density, requirements volatility, system size, or personnel effort) can be influential in determining the success of a project. In this paper we focus our attention on the issues of code product size prediction and personnel effort estimation based on measures collected during the specification and design stages of development.

Central to this paper is the contention that the components and methods comprising fuzzy logic modeling (fuzzy sets, fuzzy rules, and fuzzy inference) can be used to good effect in the management of software projects. Although more traditionally associated with hardware devices and embedded systems, fuzzy logic modeling is receiving increasing attention from the software development and project management research communities [1-4]. This has occurred largely in response to the perceived limitations of other classification and prediction methods. For instance, in contrast to many statistical and machine-learning approaches, fuzzy logic methods can be used effectively either with or without large banks of historical data. Furthermore, they have other characteristics that match those sought by project managers – for instance, they can cope with a degree of uncertainty and imprecision in classification and prediction, the models produced using fuzzy logic methods can be constructed on the basis of existing management expertise, and models can also be adjusted as new knowledge is gained. While there has been some work to date in this area [1,4-7], recent research has largely been focused on the use of fuzzy logic in retrieving ‘similar’ cases. In contrast, the emphasis here is on assessing the performance of fuzzy logic modeling in characterizing relevant variables and the relationships between them. In the case studies presented later in the paper, the accuracy of the fuzzy logic models is shown to be as good as or better than that achieved with standard linear regression.

The remainder of the paper is structured as follows: we next provide an overview of issues relating to software project management, with particular emphasis on less mature software organizations; this is followed by a description of fuzzy logic modeling in relation to software measurement; a review of the applicability of fuzzy logic modeling to software project management is presented; a description of the software environment that has been created to support the use of fuzzy logic modeling in this domain is provided; this is followed by two case studies, the first relating to software sizing and the second to effort estimation; and we close with some comments on the potential of the approach and future research opportunities.

2. SOFTWARE PROJECT MANAGEMENT

It is a truism to say that in order to manage software processes we must first know how to measure them – with such a foundation we can make predictions, generate plans, monitor activities, exert control in a sensible and reasoned manner, and finally evaluate processes and learn how they might be improved. The classifications and predictions that form part of this management activity may be constructed from formal and informal models, expert knowledge, or some suitable combination of these. Some techniques for developing such models have become well known and include function point analysis [8] and COCOMO [9] (in their various forms); alternatively regression or other data-driven models may be used as they allow the modeler greater freedom, albeit at the expense of a degree of standardization and (global) comparability.

The use of such software measurement models for prediction and classification in project management has promised much and in some cases has been very successful. However, it has also been hampered by many difficulties, not the least of which is in ensuring that the models actually capture the factors of interest and influence. Managers may possess significant knowledge about the likely relationships between factors, including variables that cannot be effectively or easily modeled in a formal or quantitative sense (e.g. team dynamics, developer fatigue, and the likely effect of new techniques and tools). These variables are those that the experienced manager takes into account implicitly when they make a 'guestimate' or instinctively round a predicted value up or down after using a formal technique. In order to improve our processes, however, we need to have repeatability and consistency in management; this generally requires the use of more formalized modeling techniques, techniques that are typically less able to take such subjective knowledge into account.

A related concern arises when experienced managers leave an organization, taking substantial amounts of project management knowledge with them. This knowledge is crucial for planning, and in many organizations, particularly those that are not mature in terms of process, such knowledge may be difficult or even impossible to replace. This information can have a significant and potentially quantifiable financial value to the organization, so attempts to retain even some of this knowledge could be crucial to its continuing viability.

To augment this expert knowledge, databases of historical data can be populated and used for model development, indexed for retrieval, and mined for trends and patterns. Data for many different software measures could be collected, including specification size, developer expertise and experience, and code quality and complexity, depending on the relationship(s) of particular interest. Less mature organizations, however, are characterized by the absence of such databases (and, for that matter, the procedures of systematic collection), making model development and subsequent calibration far less feasible. Unfortunately many modeling methods assume the availability of such a collection of records. Furthermore, the usefulness of any data that might be available is further limited by rapidly changing technology, in that new development environments, programming languages, methodologies and other factors can all make project data outdated within a short period of time [10].

In light of these difficulties, one of the most commonly used prediction 'techniques' in management practice, particularly in new or emerging software organizations, is expert estimation [11,12]. While this can be very effective, it suffers from problems of subjectivity, non-repeatability, inconsistency, and, as mentioned, vulnerability to loss of knowledge if managers leave an organization. In these circumstances what is needed is not more formalized models of metrics, nor more ways of guessing parameter values – rather, we need to find a way to *combine* expert and formal modeling without overly dulling the benefits of either [12,13].

3. FUZZY LOGIC MODELING AND SOFTWARE MEASUREMENT

(The use of fuzzy logic modeling in relation to software engineering and project management has only recently come to prominence, in spite of a small number of innovative papers published during the 1990s. We therefore include here a high-level overview of the basic concepts of fuzzy sets and fuzzy logic in this context. More detailed introductions may be found in [3] and [14]. The reader who is familiar with such material may wish to skip directly to the next section.)

Fuzzy logic is a term used to describe a set of modeling methods that enables us to accommodate the imprecision in relationships between variables. Fuzzy sets can have linguistic rather than numeric labels in order to represent the inherent uncertainty in some concepts. For instance, a system may best be described as *large* (for the concept of size), a developer may be *highly* experienced, and a program may have *very simple* structure. Figure 1 shows an illustrative group of fuzzy sets (or membership functions) derived from a project manager regarding his perspective on data model size. It provides an easily understandable view of what this expert means when he says that a system has a 'small-medium' sized data model (about 30 entities in this case), or a 'very large' data model (something over around 110 entities). A system with 60 entities is apparently what this expert would regard as exactly matching his view of a 'medium' sized data model.

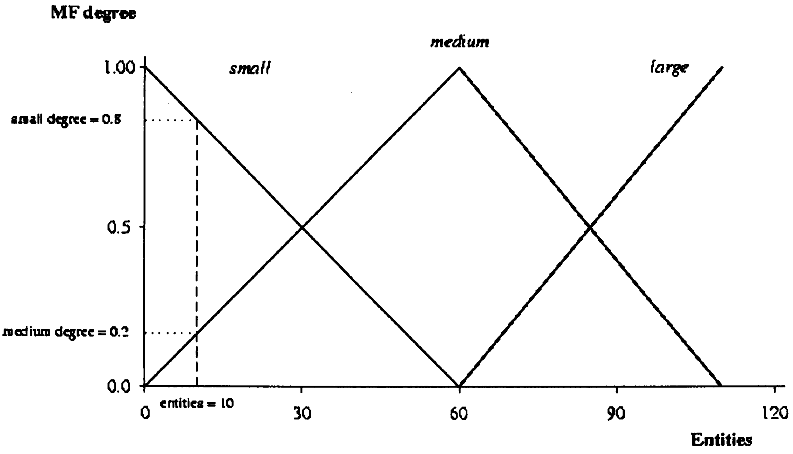


Figure 1. Example membership functions for a data model size variable.

Of course, such membership functions must first be derived for each of the concepts of interest. Common techniques used with groups of experts include polling, negotiation and voting [14,15]. Alternatively a single expert can outline their membership functions by specifying the centers of each set and then evaluating the concept at various points along the x-axis. Note that there is no reason for membership functions to be represented only as triangles. Other shapes, such as trapezoids, bell-shaped curves, and even arbitrary curves, can be used equally well. The specific shape should be chosen in order to most effectively represent the views of the expert in terms of the degree of set membership for various values within the range of interest. Or, if some data is available then it can be used to derive membership functions, usually based on some form of clustering. These data-derived functions can then be used as is, or may be employed more as a first-cut system requiring refinement by domain experts.

Membership functions define the level of mapping between values and concepts, so we have considerable freedom in how we define these functions, both directly (as in Figure 1) and implicitly – for example, adjectives such as 'about' can be defined so as to be usable for any numeric value, as in Figure 2.

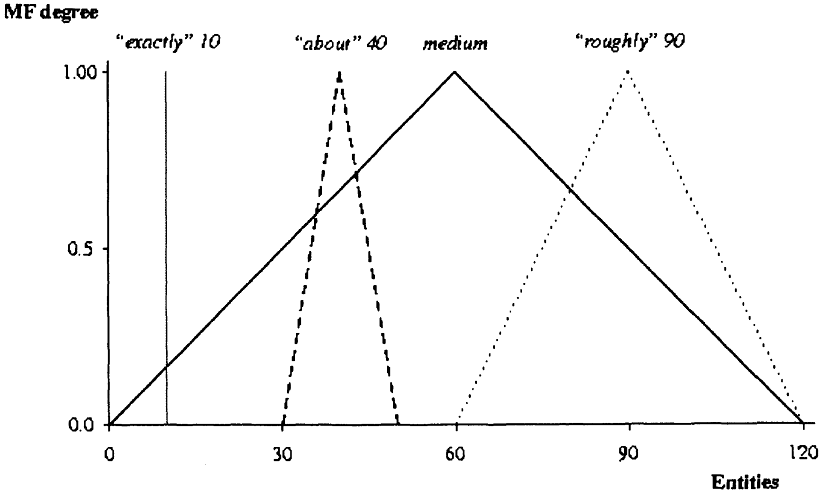


Figure 2. Different forms of linguistic membership functions.

Crucial to the fuzzy logic modeling approach is the fact that propositions are allowed to take various degrees of truth [14], so that membership of any particular fuzzy set is partial rather than exclusive. This allows a manager to describe a system as mostly ‘large’, say to a degree of 0.75 (on the scale 0 to 1) in terms of data model size, but also to some degree, say 0.40, ‘medium’. For each concept the number of membership functions can be made as high as necessary to capture the required level of granularity. A manager may choose to begin with three (to represent low, medium, and high or similar) and then extend to five, seven or more as the available information increases and accuracy demands become greater.

Given a set of membership functions, rules are then needed to relate the variables to one another as appropriate. It is here that fuzzy rules and fuzzy inference (rather than fuzzy sets) become apparent in a model, although the term *fuzzy logic* is often used to describe the entire modeling approach. Rules generally take the form of ‘these input variables with these labels/values generally lead to this output variable having this label/value’. Again, rules can be derived directly from an expert, from voting and negotiation procedures with a group, or from an existing data set.

An sample pair of simple rules relating an interface size variable (number of screens) and data model size (number of entities) with code product size (number of lines of code) could be:

```
IF screens is LOW
 AND entities is LOW
THEN size is VERY LOW

IF screens is MEDIUM
 AND entities is MEDIUM
THEN size is MEDIUM
```

The rules are fired based on the degree to which each rule's antecedents match the observed values/sets, in this case the numbers of screens and entities. Each rule then makes a contribution to the output prediction – being either a label or a numeric value – based on the degree to which its conditions are satisfied (thus interpolating between the rule points) and determined using a chosen defuzzification method. As in production systems, this process is known as inference.

The standard logical connectives and modifiers AND, OR, and NOT can be used along with bracketing to create nested rules. Rules may also be weighted, and even individual clauses can be given weights, to reflect the extent of confidence the domain experts have in a given rule or clause, with a consequential effect on the processing of the rules to determine the output label or value. There is an obvious trade-off, however, between rule sophistication (and thus the ability to make arbitrary mappings) and comprehensibility.

(Note: In this paper the term fuzzy logic modeling is used to refer to the combined use of fuzzy sets, fuzzy rules and fuzzy inference.)

4. APPLICABILITY OF FUZZY LOGIC MODELING TO SOFTWARE PROJECT MANAGEMENT

The characteristics of fuzzy logic modeling that make it suitable in principle for software project management are as follows.

Able to cope with minimal data: Since they can be based on data, knowledge, or both, fuzzy logic models can be developed with little or even no data at all (see also the following two points). This is a considerable advantage given the widely acknowledged problems encountered in data gathering in software management research and practice. It is well known that the collection of homogeneous data sets is often complicated by rapidly changing technologies and by reluctance for inter-organizational sharing of measurement data. Even within a single organization there can be considerable pressure from programmers and managers against measurement collection.

Robustness to data set characteristics: It is not uncommon for software management data sets to contain unusual or anomalous observations thus reducing the generalisability of any model derived empirically from them [16]. These atypical observations may occur for a variety of reasons – change in development practices, enhanced training for staff, or other unmeasured (perhaps unmeasurable) influences. By developing models with considerable expert involvement, where the model can be interpreted, checked and refined if needed, some of the problems of non-representative data corrupting empirically tuned models can be reduced or avoided.

Use of expert knowledge: Since fuzzy logic enables us to represent concepts as membership functions and associations between these concepts as rules we can very easily incorporate expert knowledge of such concepts and relationships in a fuzzy system. This expert knowledge is therefore naturally captured within the system, providing a means of retaining it, perhaps beyond the expert's employment. This means that organizations that are 'data-poor' but 'knowledge-rich' can still develop and use effective and locally relevant management models. This is in contrast to other data-dependent modeling methods, including statistical methods such as cluster analysis and linear regression, and connectionist methods including neural networks. Note that we are not advocating that data-driven modeling should be abandoned entirely; on the contrary, if a high quality data set is available then it can be extremely valuable in terms of providing the basis for useful models. Rather, we are simply suggesting that total reliance on empirically derived models may not always provide an optimal solution, particularly in circumstances where the data set is small, incomplete or limited in other ways. A modeling method that sensibly *incorporates* the views of experienced personnel and combines this dynamically with appropriate data as it becomes available would seem to the most preferable approach. This seems all the more reasonable when it is considered that expert estimation is still prevalent among many software organizations [11]. (As a result, our software toolset (described later in the paper) can accommodate both data and expert knowledge as appropriate for each relationship being modeled.)

Able to cope with uncertainty: The terms 'prediction', 'estimate' and the like are generally understood to indicate a forecast value. In terms of good business practice it is normally in our interests to provide estimates that are as close as possible to the values actually achieved or incurred. That said, there should also be a degree of acceptance that there is some chance that the actual value will not match the estimate made. Such an outcome becomes increasingly likely when we have only minimal information on which to base our prediction. Where this is the case, however, our estimate may be made more accurate as we move through the process and more information becomes available. In many industries such a scenario operates quite effectively. In the software industry, however, the provision of accurate estimates early in the development process is much more difficult.

Requirements are volatile, processes vary, personnel are transient, technology moves rapidly – change is endemic. Under these circumstances it would seem sensible to use modeling methods that enable us to formulate estimates with some form of associated confidence factor, along with an assessment of the source and level of risk that the estimate will not correspond to the actual value. Early estimates are clearly needed – modeling methods that enable early 'ball park' predictions to be developed, but that deliver them to both manager and client with appropriate qualification either as a label or as a likely range of values, would appear to be an acceptable compromise. Methods should also be sufficiently flexible to enable the incorporation of contingencies based on sound organizational knowledge [17,18]. Fuzzy logic models facilitate the provision of confidence-weighted predictions as well as the inclusion of relevant contingencies.

Varying granularity: A further related advantage of fuzzy logic models, but one that is less frequently cited, is the ability to change the level of granularity in the inputs and outputs without changing the underlying rule set [19]. For example, Figure 3 shows the inputs to and outputs from a rule set at different stages of a software development process. During requirements analysis the inputs to and outputs from the model are linguistic, they become fuzzy numbers during design, and evolve into exact values in later stages, when more is known. This increasing specificity can be easily accommodated using the same fuzzy logic model so there are no problems with inconsistent models when moving from phase to phase. In short, we can design a fuzzy logic model so that input granularity reflects what we actually know, and output granularity reflects what we need to know.

'Free' modeling capability: As opposed to more formalized algorithmic techniques, such as function point analysis and linear regression, fuzzy logic models can include any variables at all and the inference process can easily account for non-linearity and interactions in relationships.

Easily learned and transparent: Relatively speaking, fuzzy logic modeling is considerably easier to understand and use than many statistical and neural network techniques. While there are several important underlying mathematical principles, the technique lends itself to use by novices as well as experts [14,20]. Fuzzy logic models are also completely transparent in that the mapping from inputs to outputs can be examined, analyzed, and revised where necessary. This can be especially important in terms of a model's acceptance by the personnel affected by its use. An ideal modeling method should strike a balance between effectiveness in terms of accuracy and consistency on the one hand and simplicity in calculation and application on the other.

In spite of what appears to be sound rationale for the application of fuzzy logic in software engineering and management, work to date in this area has not been extensive. One of the earliest papers to address this general area reported on the use of fuzzy sets in assessing software correctness [21]. In 1994 Kumar *et al.* [22] provided an excellent introduction to the potential of fuzzy logic and neural networks in software engineering project management, but perhaps because it appeared in an artificial intelligence journal rather than mainstream software engineering literature the ideas relating to fuzzy logic were not widely adopted in this domain. The same may be said of the work of Shipley *et al.* [2], which was reported in the engineering and technology management literature. In the wider context of systems management de Ru and Eloff [23] used a simulated case study to illustrate the potential of fuzzy logic modeling in computer security risk assessment. Looking specifically at the synthesis of fuzzy logic and software management, Khoshgoftaar and Allen and their colleagues have undertaken some of the more prominent work in this area, although their focus had tended to be on other modeling methods, including neural networks, genetic programming and decision trees [24,25]. Ebert [26] assessed the comparative performance of a similar set of methods in determining error-prone modules in a large telecommunication system. More recently, Idri, Abran and others have used fuzzy logic in relation to software development effort prediction using a fuzzy variant of the COCOMO model [1] and in creating more forgiving methods for the retrieval of analogous cases in software project management [5,6].

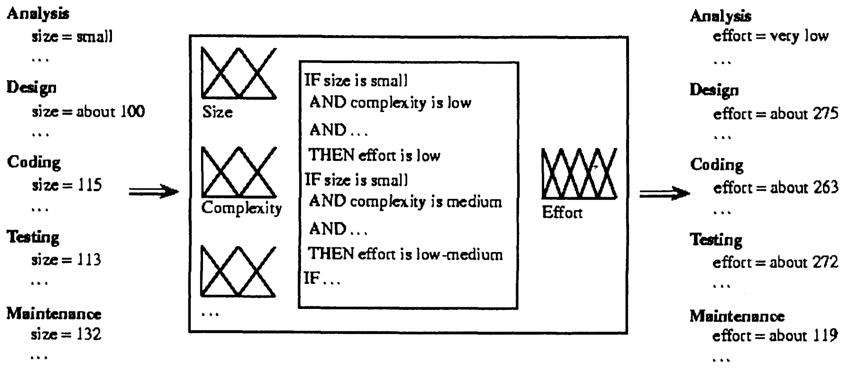


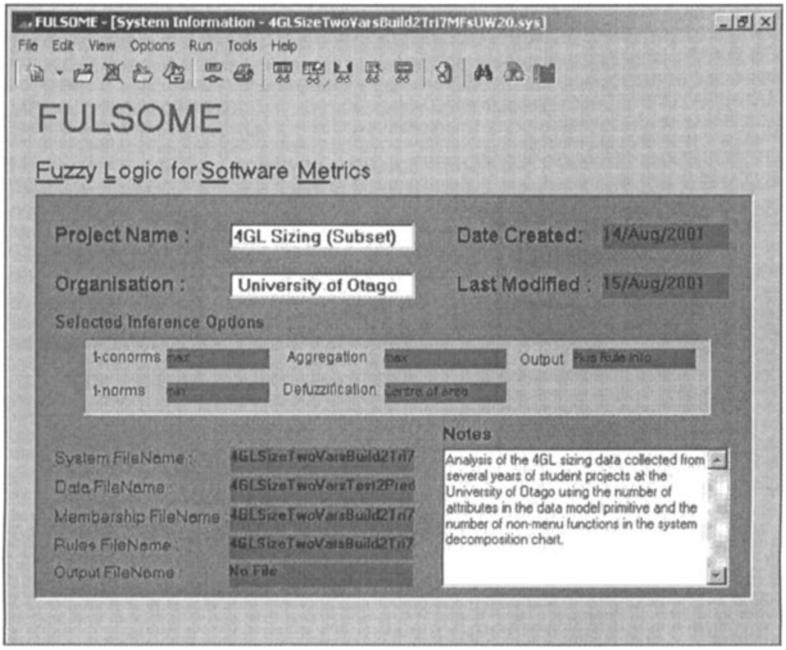
Figure 3. Changes in input and output granularity throughout the development process.

5. THE FUZZYMANAGER TOOLSET

Since there is limited benefit in proposing solutions to problems without providing developers and managers with the means to implement those solutions [27], we have developed a freely available fuzzy logic modeling toolset called FUZZYMANAGER [20]. The toolset consists of two modules, FULSOME (FUzzy Logic for SOftware MEtrics) and CLUESOME (CLUster Extraction for SOftware MEtrics).

The FULSOME module (Figure 4a-d) supports the graphical creation and refinement of membership functions (Figure 4b) and rule bases (Figure 4c) using expert knowledge, as well as tracing the inference process at the observation, rule, and rule clause levels (Figure 4d). This latter component is important from the perspective of credibility, delivered through model transparency, and robustness, in that any anomalies in the inference process can be identified and corrected where appropriate.

The FULSOME process can be augmented through the incorporation of relevant data observations. In fact, an organization that has systematically collected data concerning a particular relationship but that does not understand the ‘rules’ governing that relationship can use the CLUESOME module to derive membership functions and rule bases from that data (either together or separately), using a simple form of fuzzy c-means clustering. Two separate algorithms are used, one for membership functions and another for rules. Some approaches to obtaining fuzzy logic systems from data use highly complex algorithms that produce extremely effective mappings between inputs and outputs. Here we have used a simpler method that generally produces smaller and more intuitively acceptable rule sets, given that there is a trade-off between understandability and accuracy in such sets. Just as importantly, project managers in our collaborative partner organizations have understood the algorithms without difficulty. The extracted membership functions and rules can then be imported back into FULSOME for further processing and possible refinement. (The case studies discussed later in the paper both employ a data-driven approach to model development, to illustrate the use of both CLUESOME and FULSOME modules.)



FULSOME - [System Information - 4GLSizeTwoVarsBuild2Tri7MFsUW20.sys]

FULSOME

Fuzzy Logic for Software Metrics

Project Name : 4GL Sizing (Subset)

Date Created: 14/Aug/2001

Organisation : University of Otago

Last Modified : 15/Aug/2001

Selected Inference Options

t-conorms max

Aggregation max

Output Plus Rule info

t-norms min

Defuzzification Centre of area

System FileName : 4GLSizeTwoVarsBuild2Tri7

Data FileName : 4GLSizeTwoVarsTest2Pred

Membership FileName : 4GLSizeTwoVarsBuild2Tri7

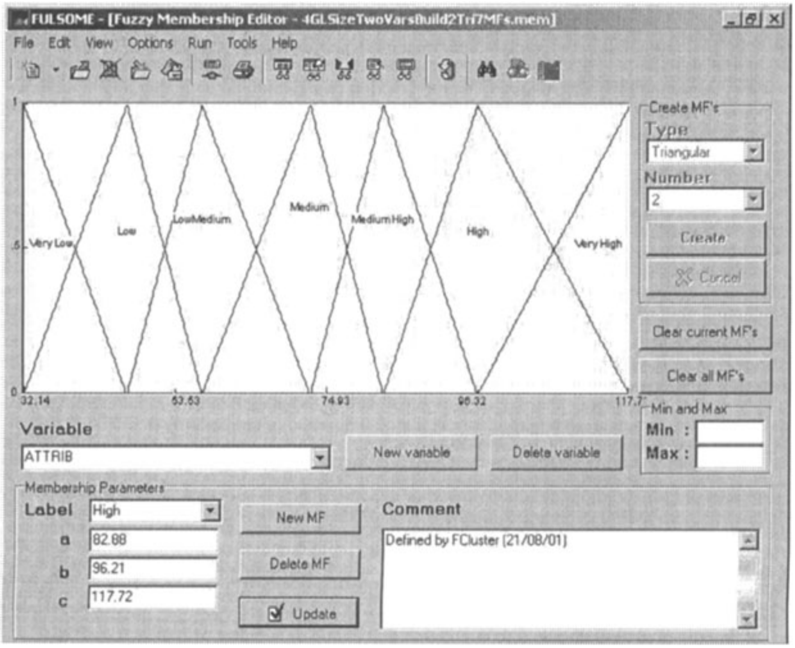
Rules FileName : 4GLSizeTwoVarsBuild2Tri7

Output FileName : No File

Notes

Analysis of the 4GL sizing data collected from several years of student projects at the University of Otago using the number of attributes in the data model primitive and the number of non-menu functions in the system decomposition chart.

Figure 4a. FULSOME system screen.



FULSOME - [Fuzzy Membership Editor - 4GLSizeTwoVarsBuild2Tri7MFs.mem]

Variable: ATTRIB

Membership Parameters

Label: High

a: 82.88

b: 96.21

c: 117.72

Comment

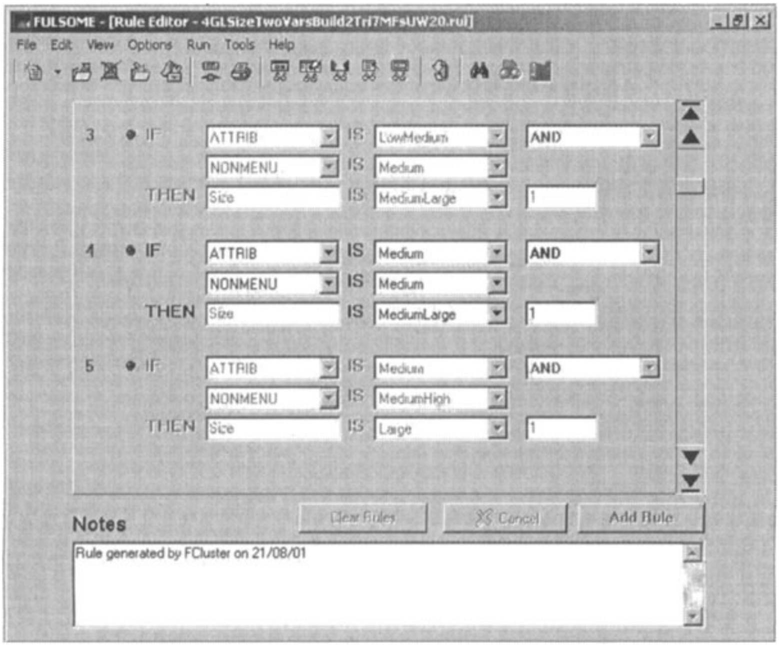
Defined by FCluster [21/08/01]

Create MF's

Type: Triangular

Number: 2

Figure 4b. FULSOME membership function screen.


FULSOME - [Rule Editor - 4GLSizeTwoVarsBuild2Tri7MFsUW20.rul]
File Edit View Options Run Tools Help
Clear Rules
Cancel
Add Rule

3 ● IF ATTRIB IS LowMedium AND
NONMENU IS Medium
THEN Size IS MediumLarge 1

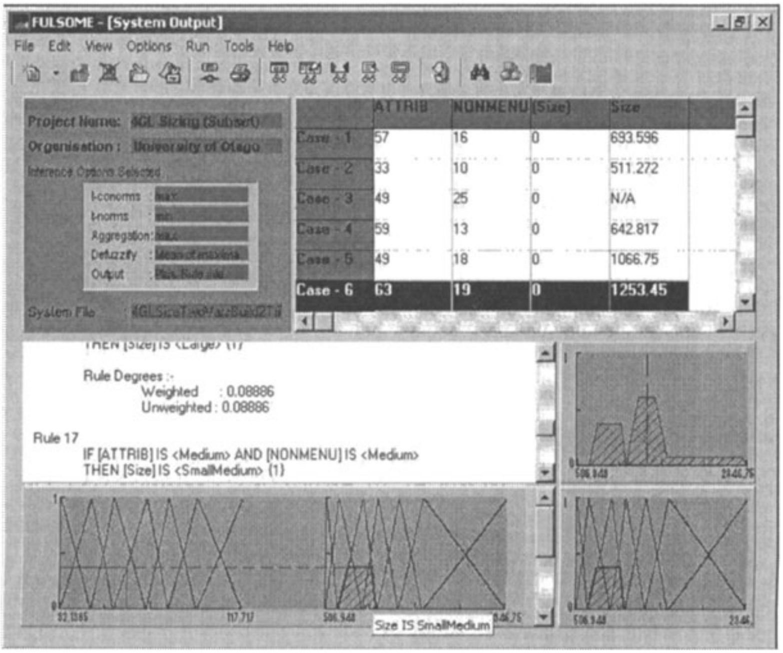
4 ● IF ATTRIB IS Medium AND
NONMENU IS Medium
THEN Size IS MediumLarge 1

5 ● IF ATTRIB IS Medium AND
NONMENU IS MediumHigh
THEN Size IS Large 1

Notes

Rule generated by FCluster on 21/08/01

Figure 4c. FULSOME rules screen.



Project Name: 4GL Sizing (Subset)
Organisation : University of Otago
Inference Options Selected
t-conorms : max
t-norms : min
Aggregation: max
Defuzzify : Mean of maxima
Output : Plus Rule ...
System File : 4GLSizeTwoVarsBuild2Tri

	ATTRIB	NONMENU	(Size)	Size
Case - 1	57	16	0	693.596
Case - 2	33	10	0	511.272
Case - 3	49	25	0	N/A
Case - 4	59	13	0	642.817
Case - 5	49	18	0	1066.75
Case - 6	63	19	0	1253.45

THEN [Size] IS <Large> (1)

Rule Degrees :-
Weighted : 0.08886
Unweighted : 0.08886

Rule 17
IF [ATTRIB] IS <Medium> AND [NONMENU] IS <Medium>
THEN [Size] IS <SmallMedium> (1)

Figure 4d. FULSOME output screen.

The following steps are performed in the membership function extraction algorithm: the user selects the number of membership functions and their shapes for each of the variables of interest; the clustering algorithm finds the centers of the clusters for each variable; each cluster center is used for the center of a membership function with appropriate parameters used to connect it to its adjacent functions (if any). This process is more formally expressed as follows:

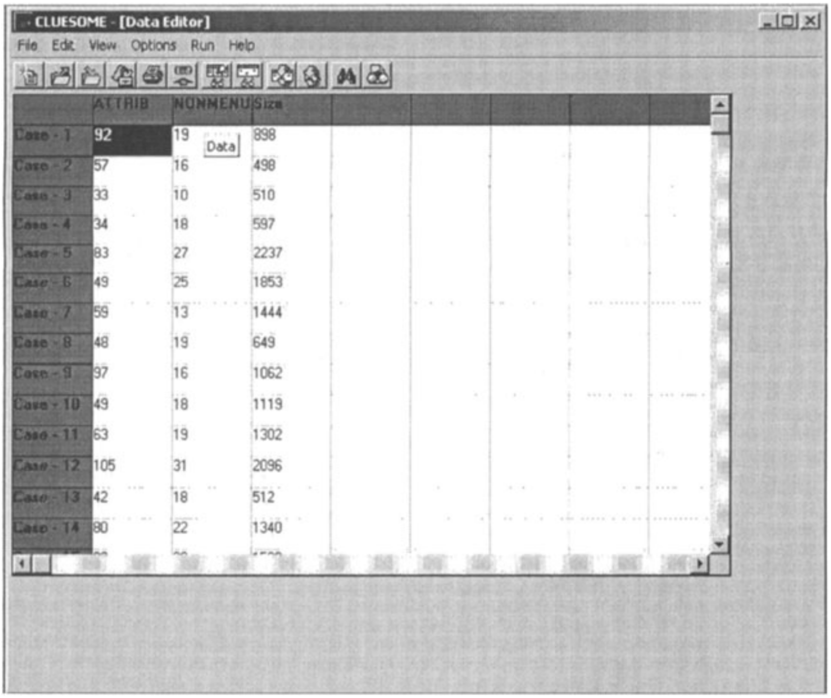
1. select an appropriate mathematically defined function for the membership functions of the variable of interest i, say $f_i(x)$
2. select the number of membership functions that are desired for that particular variable, m_i
3. call each of the m_i functions $f_{ij}([x])$ where $j = 1 \ldots m_i$ and $[x]$ is an array of parameters defining that particular function (normally a center and width parameter are defined)
4. using one-dimensional fuzzy c-means clustering on the available data set find the m_i cluster centers, c_{ij}
5. sort the cluster centers c_{ij} into monotonic (generally ascending) order for the given i
6. set the membership function center for f_{ij}, generally represented as one of the parameters in the array $[x]$, to the cluster center c_{ij}
7. set the membership function widths for f_{ij} in $[x]$ such that $\sum^{mi}_{n=1} f_{in}([c_{in}, \ldots]) = 1$, or as close as possible for the chosen $f(x)$ where this cannot be achieved exactly (for example for triangular membership functions each function can be defined using three points a, b and c where a is the center of the next smaller function and c is the center of the next larger function).

In the rule extraction algorithm the following steps are undertaken: the user selects the number of rules desired; the clustering algorithm finds the cluster centers; the membership functions that are mostly highly activated for each variable are used as the antecedents and consequents of each rule. Optionally, the label activation degrees can be used to produce rule weights, and rules with the same set of labels can be combined to produce a smaller rule base. More formally, this can be described as follows:

1. start with known membership functions $f_{ij}([x])$ for all variables, both input and output, where j represents the number of functions for variable i and $[x]$ is the set of parameters for the particular family of function curves
2. select the number of clusters k (which represents the number of rules involving the k-1 independent variables to estimate the single output variable)
3. perform fuzzy c-means clustering to find the centers (i dimensional) for each cluster

4. for each cluster *k* with center c_k
    a. determine the *k*th rule to have the antecedents and consequent f_{ij} for each variable *i* where $f_{ij}(c_k)$ is maximized over all *j*
    b. weight the rule, possibly as $\prod^{i}_{n=1} f_{ij}(c_k)$ or $\sum^{i}_{n=1} f_{ij}(c_k)$
5. combine rules with the same antecedents and consequents, either summing, multiplying, or bounded summing rule weights together
6. (optionally) ratio scale all weights so that the mean weight is equal to 1.0 to aid interpretability.

The two components of the CLUESOME module, the data entry/edit screen and the cluster view screen, are shown in Figure 5a-b.



	ATTRIB	NONMENU	Size
Case - 1	92	19	898
Case - 2	57	16	498
Case - 3	33	10	510
Case - 4	34	18	597
Case - 5	83	27	2237
Case - 6	49	25	1853
Case - 7	59	13	1444
Case - 8	48	19	649
Case - 9	97	16	1062
Case - 10	49	18	1119
Case - 11	63	19	1302
Case - 12	105	31	2096
Case - 13	42	18	512
Case - 14	80	22	1340

Figure 5a. CLUESOME data entry/edit screen.

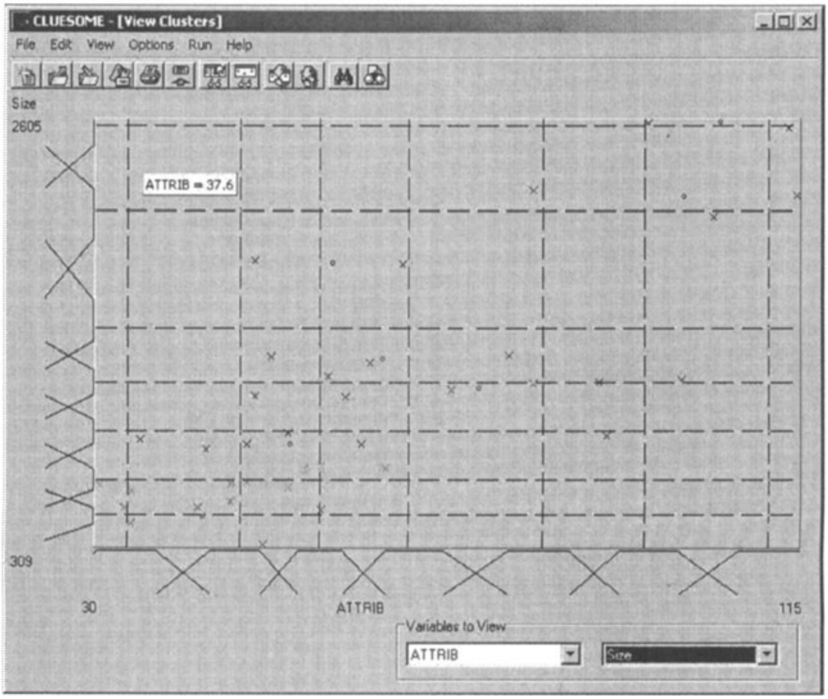


Figure 5b. CLUESOME cluster screen.

6. CASE STUDY 1 – SOFTWARE SIZING

To illustrate the viability of fuzzy logic modeling in software project management we conducted a case study utilizing a previously published software engineering data set concerned with code product size prediction for 4GL systems (see [28]). Data had been routinely collected over a period of five years relating to the development of small- to medium-sized 4GL transaction processing systems by groups of senior students at the University of Otago in New Zealand. This led to the availability of a data set comprising 70 observations, each incorporating measures relating to the size of the data model and the functional decomposition chart from the requirements specification and a count of the number of source statements of 4GL code that comprised the delivered system. Although we did have some high-level knowledge of the systems, and of the personnel involved in their development and management, these individuals were no longer part of the faculty. We therefore decided to treat this case study as one that might match the scenario in industry whereby experienced development and management personnel had left the organization, taking with them their knowledge of relevant size prediction models. This implied the use of CLUESOME in order to build a first-cut fuzzy system from the existing data set.

Given that our aim here was principally to illustrate the feasibility of fuzzy logic modeling in the context of software project management the actual data set was not especially important – any systematically collected software management data set could have been used. This set was chosen, however, because we knew something about the systems and had previously analyzed the data using standard statistical methods. This analysis, reported in [28] along with the full data set, indicated that an effective predictive model of 4GL system size could be constructed using linear regression and incorporating measures of the number of attributes in the system data model and the number of non-menu system functions (in other words, data entry/edit screens and periodic reports) depicted in the functional hierarchy as predictor variables. In order to provide a basis for comparison in terms of model performance we decided to use this model as a benchmark and as a starting point for our fuzzy logic model construction. Thus our model was to consist of two predictor variables – the number of attributes in the system data model (ATTRIB) and the number of non-menu functions in the functional hierarchy (NONMENU) – and one independent variable – the size of the implemented system in 4GL source statements (Size).

In order to develop useful predictive models it is common practice to split any available data set into two parts – the first is used to build a 'best fit' model, and the second is used to assess that model's accuracy on an unseen collection of observations. We also chose to follow this convention. Moreover, previous research has indicated that model construction using empirical analysis of software engineering data sets can be influenced significantly by the samples used in the construction process [29,30]. In order to lessen the possible bias of a single sample, we therefore took two separate random samples of twenty observations as our test samples, leaving two samples of fifty observations for the model building process.

The CLUESOME module of FUZZYMANAGER was used to generate membership functions and rule sets for each of the two build samples. Several membership function structures were considered, including bell, trapezoidal and triangular shaped sets numbering five or seven for each of the three variables. The fit of each model was considered in terms of six measures of accuracy (where the relative error of each prediction equals the actual value less the predicted value all divided by the actual value): mean magnitude of relative error (MMRE), median magnitude of relative error (MedMRE), the proportions of predictions falling within 20% and 30% of the actual values (pred(20) and pred(30) respectively), and two indicators of absolute error, the sum of the absolute difference (SumAbsDiff), and the median absolute difference (MedAbsDiff). Whilst the MMRE, MedMRE and pred indicators have been widely used in empirical software engineering research opinion is divided on their appropriateness, hence the inclusion of the simpler absolute error measures.

It should be noted that we did not expend a large amount of effort optimizing the accuracy of our build models, for three reasons. One, high accuracy on a build sample does not necessarily translate to similar or better performance on an independent test sample, where it is actually desired; two, we may over-fit the model to the build sample at the expense of model generalisability; and three, we wanted to assess how well a reasonably rudimentary fuzzy model performed against standard statistical methods. To this end CLUESOME produces straightforward outputs – it simply looks for important rules based on all of the predictor variables (antecedents), combined using only the AND connective. Thus it can be considered that the generated models are at the low end of the spectrum in terms of sophistication (and therefore complexity). This illustrates our intent that CLUESOME provide easily understandable fuzzy systems as a starting point for expert refinement, rather than highly accurate but potentially very complex systems.

For both build samples it was found that seven rather than five membership functions provided the most accurate models. For the ATTRIB and NONMENU variables we used the labels VeryLow, Low, LowMedium, Medium, MediumHigh, High and VeryHigh to indicate the spread of values. A slightly different set of labels was used for Size: VerySmall, Small, SmallMedium, Medium, MediumLarge, Large and VeryLarge. Trapezoidal sets proved to be slightly more effective for build sample one, whereas triangular sets performed most effectively for the second build sample, although the difference in performance for both samples was not large. In both cases, however, bell-shaped curves did not result in usefully accurate models.

Another difference resulting from the use of two different build samples was the number of extracted rules to provide the most accurate model. We evaluated just two sizes of rule set, at fifteen and twenty rules respectively. The most accurate model for build sample one comprised fifteen rules whereas the best-performing model for build sample two was made up of twenty rules. Again, we could have evaluated many other rule set sizes in order to possibly achieve further incremental improvement, but for the reasons stated above we maintained a reasonably simple build and assessment strategy.

The form of the systems as generated and imported into FULSOME is illustrated in the screen shots shown in figure 4. Figure 4b shows the seven triangular membership functions created for the ATTRIB variable from build sample two. A selection of the associated set of rules is shown in figure 4c. If no data had been available, but the organization had substantial experiential knowledge of the important concepts (variables) and their relationships, these same components of the system could have been created and edited directly in FULSOME.

The selected models were then applied in a predictive capacity in FULSOME against their corresponding test samples. Input values for the two independent variables were mapped onto the output functions using the rule sets, and point estimates of Size were inferred for each observation (although linguistic labels or ranges could have been chosen instead). Table 1 provides a summary of the accuracy of the two fuzzy models. For the purposes of comparison, these results are shown alongside the performance of linear regression models generated for each of the build samples and then applied against the same test samples as their fuzzy counterparts.

	Regression Model 1	Fuzzy Model 1	Regression Model 2	Fuzzy Model 2
MMRE	0.22	0.17	0.31	0.31
MedMRE	0.16	0.10	0.17	0.17
pred(20)	55%	50%	55%	45%
pred(30)	65%	55%	65%	55%
No pred	0%	30%	0%	10%
pred(20)s	55%	71%	55%	50%
pred(30)s	65%	79%	65%	61%
SumAbsDiff	4463	2367	5557	4992
MedAbsDiff	175	97	198	146

Table 1. Comparative performance of regression and fuzzy logic models for the two test samples (case study 1).

It can be seen in Table 1 that in terms of the MMRE, MedMRE, SumAbsDiff and MedAbsDiff measures the two fuzzy logic models performed either as well as or better than the corresponding regression models. Using the pred(20) and pred(30) indicators, however, the regression models were superior for both samples. These results are confounded, however, by the fact that in both cases the fuzzy logic model failed to provide predictions for some of the observations in the test samples. This occurred for six of the twenty observations in sample one and two of the twenty cases in sample two (indicated as 30% and 10% in the 'No pred' line of Table 1). These cases made no contribution to the MMRE and MedMRE calculations leading to what might be seen as optimistic values for the fuzzy logic models. On the other hand, the pred values for the fuzzy logic models are pessimistic as they were calculated over the entire set of twenty observations even though for some no estimates were made. In order to provide further insight in terms of performance we have also included measures of pred accuracy over the subset of observations for which an estimate was produced, denoted as pred(*l*)s in Table 1. This indicates that although fuzzy model 1 only produced an estimate for fourteen of the twenty cases, the accuracy of those predictions was good, and superior to that achieved using linear regression.

The fact that estimates were not produced in some cases clearly warrants some discussion. The reason is relatively straightforward – the input values in these observations, or more correctly the fuzzy sets to which the input values belonged, failed to match any combination of those specified in the rules comprising the respective rule sets. Not surprisingly, this was a more significant issue for model one that comprised just fifteen rules when compared to the twenty-rule set available to model two. It illustrates well the potential impact of both sampling and rule set size selection on the efficacy of the resulting models. This outcome could be interpreted with some concern – an organization receiving estimates for only some of their activities may consequently feel that the modeling method is inadequate, particularly if it is considered that linear regression models, once constructed, can produce an estimate for all input values. We may equally come to an alternative, more positive interpretation, however. It may be considered a good thing that the model does not attempt to provide a prediction when there is no sound basis for doing so, particularly given that we are already incorporating a degree of imprecision through the use of fuzzy sets. If indeed no rules were fired then this would imply that there is insufficient information to enable an accurate prediction to be made, just as extrapolation of a regression model outside the bounds of the data on which it was built is risky (although not prohibited).

On the basis of the results presented above, using what were relatively rudimentary fuzzy logic models, it can be concluded that the approach has the capacity to perform at least as well as standard linear regression modeling in some cases. In order to assess whether slightly more advanced analysis could lead to further improvement in accuracy we refined the two selected models by using CLUESOME to generate weighted rule sets for each sample, rather than using the default weight of 1 for every rule. This enables a measure of confidence to be associated with each rule based on the strength of the clustering that led to its derivation (as in step 4(b) of the rule extraction algorithm described above). Whilst no improvement in performance was obtained for fuzzy logic model 1 as a result of this further analysis, gains were made for fuzzy logic model 2 for all of the accuracy measures employed (see Table 2).

	Regression Model 1	Fuzzy Model 1	Regression Model 2	Weighted Fuzzy Model 2
MMRE	0.22	0.17	0.31	0.29
MedMRE	0.16	0.10	0.17	0.13
pred(20)	55%	50%	55%	50%
pred(30)	65%	55%	65%	60%
No pred	0%	30%	0%	10%
pred(20)s	55%	71%	55%	56%
pred(30)s	65%	79%	65%	67%
SumAbsDiff	4463	2367	5557	4005
MedAbsDiff	175	97	198	123

Table 2. Comparative performance of regression and weighted fuzzy logic models (case study 1).

In the final phase of case study 1 we simulated the real-world scenario whereby a model would be refined using expert analysis. As we had some knowledge of the systems and their construction, there was some basis for such an adjustment. We were by no means experts, however, so our motivation was as much to improve model accuracy as to produce truly representative models. In fuzzy model 1 we made minor changes to the lower bound parameters of the Low and High fuzzy sets for ATTRIB. (Three of the six observations for which no prediction had been made had relatively low values for ATTRIB and one had a high value for the same parameter.) We also made one adjustment to the Size label, from SmallMedium to Small, in a single rule in the fifteen-rule set. These changes had a significant impact on both the coverage and accuracy of fuzzy model 1, as shown in Table 3. For the sample two model the addition of a rule dealing with the prediction of large systems, along with minor refinements to the Low and Medium fuzzy sets of the ATTRIB membership function, led to the provision of reasonably accurate predictions for all twenty observations.

Examination of the data also revealed an outlier value in test sample two, with an MRE more than twice that of its closest rival, at 1.59 and 1.80 for the regression and fuzzy logic models respectively. The degree of influence of this observation had been noted in the previous analysis [28]: “This project was found to be the smallest of the entire sample at just 309 lines of source code. On further investigation, it was found that the system had been developed using the maximum of default settings and generated code, with very little programmer adaptation to customize the functionality and user interface employed. Although admittedly *unusual*, this did not make the project *invalid* in terms of the study - thus the observation was left in.” Its impact is evident in the inflated MMRE values for both sample two models.

	Regression Model 1	Amended Fuzzy Model 1	Regression Model 2	Amended Fuzzy Model 2
MMRE	0.22	0.18	0.31	0.28
MedMRE	0.16	0.10	0.17	0.12
pred(20)	55%	60%	55%	60%
pred(30)	65%	75%	65%	70%
No pred	0%	10%	0%	0%
pred(20)s	55%	67%	55%	60%
pred(30)s	65%	83%	65%	70%
SumAbsDiff	4463	2831	5557	4206
MedAbsDiff	175	90	198	110

Table 3. Comparative performance of regression and 'expert' amended fuzzy logic models (case study 1).

7. CASE STUDY 2 – SOFTWARE DEVELOPMENT EFFORT

Our second case study deals with development effort prediction, and is based on a data set published by Miyazaki *et al.* [31]. Development effort in person-months (MM) had been recorded for 48 systems, along with values of several predictor variables, including the numbers of screens required (SCRN), forms to be produced (FORM) and files to be accessed (FILE). Examination of the data set revealed a significant outlier observation, illustrated clearly by the fact that the mean effort value across the sample was 87 person-months whereas the outlier observation had an effort value of 1586 person-months. As a result this observation was removed from the analysis, leaving a set of 47 observations for further investigation. Further preliminary examination indicated that models employing the FORM and FILE variables would provide the most effective predictive capability.

As in the previous case study we split the data set into build and test sub-samples, comprising 30 and 17 observations respectively. We also repeated the sampling process so that we were able to undertake two separate analyses on the data set. The CLUESOME module of FUZZYMANAGER was employed to generate first-cut fuzzy models from the build sub-samples before considering whether the resulting classes and rules might be improved with refinement. In this case study the number of membership functions involved in the best-performing models was different for the two build samples, at seven and five respectively. Triangular functions proved to be the most effective in both cases, however, and twenty rules proved to be more useful than fifteen for both sub-samples. Applying the models as constructed to the test sub-samples of seventeen observations and comparing this to regression analyses resulted in performance as shown in Table 4.

	Regression Model 1	Fuzzy Model 1	Regression Model 2	Fuzzy Model 2
MMRE	1.10	1.04	0.52	0.59
MedMRE	0.92	0.68	0.43	0.33
pred(20)	18%	12%	24%	18%
pred(30)	18%	12%	35%	35%
No pred	0%	18%	0%	24%
pred(20)s	18%	14%	24%	23%
pred(30)s	18%	14%	35%	46%
SumAbsDiff	521	498	719	226
MedAbsDiff	29	20	24	17

Table 4. Comparative performance of regression and fuzzy logic models for the two test samples (case study 2).

The results presented in Table 4 generally support the use of fuzzy logic modeling over regression analysis, but again this is at least in part a function of the fact that the fuzzy models did not produce predictions for three and four of the 17 test observations respectively. This has a particularly significant impact on the SumAbsDiff value for the second sample. We also employed CLUESOME to produce weighted rule sets for the two fuzzy models to increase the influence of the rules in which confidence was high. In both cases this led to small but useful improvements in performance for most of the assessment indicators (see Table 5).

	Regression Model 1	Weighted Fuzzy Model 1	Regression Model 2	Weighted Fuzzy Model 2
MMRE	1.10	1.01	0.52	0.56
MedMRE	0.92	0.69	0.43	0.36
pred(20)	18%	12%	24%	24%
pred(30)	18%	18%	35%	35%
No pred	0%	18%	0%	24%
pred(20)s	18%	14%	24%	31%
pred(30)s	18%	21%	35%	46%
SumAbsDiff	521	486	719	217
MedAbsDiff	29	20	24	14

Table 5. Comparative performance of regression and weighted fuzzy logic models (case study 2).

As we had not been involved in the development of the systems at the center of case study 2, or in the associated data collection exercise, we were not in a position to amend the classes and/or rules on the basis of 'expert' knowledge. In order to provide a more direct comparison of regression analysis and fuzzy modeling for this case study, however, we did amend the two fuzzy systems so that coverage of the observations was increased. This involved the inclusion of one further rule for the first sample, and the addition of two membership functions for the second sample, in order to deal with the larger observations that had by chance only been allocated to the test subsets. Assessment of performance of the amended systems relative to their regression counterparts is reported in Table 6.

	Regression Model 1	Amended Fuzzy Model 1	Regression Model 2	Amended Fuzzy Model 2
MMRE	1.10	0.75	0.52	0.61
MedMRE	0.92	0.34	0.43	0.38
pred(20)	18%	29%	24%	29%
pred(30)	18%	47%	35%	35%
No pred	0%	0%	0%	6%
pred(20)s	18%	29%	24%	31%
pred(30)s	18%	47%	35%	38%
SumAbsDiff	521	330	719	436
MedAbsDiff	29	13	24	18

Table 6. Comparative performance of regression and amended fuzzy logic models (case study 2).

In concluding the case studies it is evident that the use of fuzzy logic modeling can lead to the provision of useful predictions in software project management. Furthermore, in the first case study in particular the fuzzy logic modeling approach, combining existing data with a small degree of 'expert' knowledge, led to models that outperformed their linear regression alternatives. While it is true that we did not attempt to refine the regression models to any great extent, we feel that this is justified as our focus was primarily on assessing the feasibility of the fuzzy logic modeling approach.

8. SUMMARY, CONCLUSIONS AND FUTURE WORK

In this paper we have illustrated that fuzzy logic modeling appears to have potential applicability in the domain of software project management. We have shown that fuzzy logic modeling can be used to effectively represent software project management relationships and, in our case studies, to do so accurately when compared to more commonly used linear regression analysis methods. When this level of performance is considered alongside the additional benefits of a fuzzy logic approach – ability to cope with minimal data where necessary, robustness to data set characteristics, ability to incorporate expert knowledge, flexibility to cope with uncertainty and changing granularity, 'free' modeling capability, and model/process transparency – the potential of such an approach looks promising indeed, particularly for organizations with relatively immature or emerging management processes.

The work reported here illustrates just one application of fuzzy logic modeling, that of simple data-driven model development followed by minor 'expert' amendment. The underlying techniques and their implementation in the FUZZYMANAGER toolset could also be used to support direct elicitation and modeling of knowledge, or to fuzzify concepts and/or data for further processing using other modeling methods. While our case studies illustrated their use in prediction, they could be applied in a similar manner for classification tasks. They could also be used at other levels of software management – an organization's project portfolio risk profile could be modeled using such an approach, as could the internal structure and defect profile of low-level code modules.

In the light of the specific results obtained in the case studies we intend to examine the impact of sampling on the coverage and stability of extracted rule sets. It may be possible to generate a 'core' set of rules that hold irrespective of the sample used in their derivation. Stratified rather than random sampling may well result in more generally effective models. We have also placed responsibility for specifying some of the model parameters over to the manager, including the number of membership functions and the number of rules generated. The values chosen clearly have an impact on the resulting models – an extreme solution would be to have sufficient membership functions and rules to create a one-to-one mapping from input to output values. However, this is not knowledge in any real sense, and the models would likely be difficult to understand, analyze and revise. Our case study samples, with either fifteen or twenty-one rules, are probably of reasonable size in that they are small enough to be understood, but large enough to give good coverage and accuracy.

In terms of other future work there is a pressing need to evaluate the approach in an industry setting. While the case studies have illustrated the potential of the approach this needs to be verified against real-world and larger scale software management challenges, perhaps in the areas of application mentioned above. This will almost certainly result in the development of more complex rule sets – it remains to be seen whether such rule sets remains intuitively appealing to managers. We also intend to assess the effectiveness of various processes and methods for knowledge elicitation, so that the most effective representations possible can be developed.

ACKNOWLEDGMENTS

The authors would like to thank James Calvert, Richard Kilgour and Martin Rutherford for their work on the FUZZYMANAGER toolset, and Martin Shepperd for valuable discussions and comments on an early draft of this paper. This work was supported by the University of Otago, Auckland University of Technology and Bournemouth University.

REFERENCES

[1] Idri, A., Abran, A. and Kjiri, L. "COCOMO Cost Model Using Fuzzy Logic", In *Proc. 7th Intl Conference on Fuzzy Theory and Technology*. New Jersey, 2000

[2] Shipley, M. F., de Korvin, A. and Omer, K. "BIFPET Methodology Versus PERT in Project Management: Fuzzy Probability Instead of the Beta Distribution". *Journal of Engineering and Technology Management* 14, 1997, 49-65

[3] Pedrycz, W. and Peters, J.F. (eds) *Computational Intelligence in Software Engineering*. World Scientific: Singapore, 1998

[4] Gray, A.R. and MacDonell, S.G. "Applications of Fuzzy Logic to Software Metric Models for Development Effort Estimation", In *Proc. 1997 Annual Meeting of the North American Fuzzy Information Processing Society* - NAFIPS. IEEE Computer Society Press, 1997, 394-399

[5] Idri, A. and Abran, A. "Towards a Fuzzy Logic Based Measures for Software Projects Similarity", In *Proc 6th MCSEAI'2000 Maghrebian Conference on Computer Sciences*. Fez, Morocco, 2000

[6] Idri, A. and Abran, A. "A Fuzzy Logic Based Set of Measures for Software Project Similarity: Validation and Possible Improvements", In *Proc 7th Intl Symposium on Software Metrics*, London, IEEE Computer Society Press, 2001, 85-96

[7] Gray, A.R. and MacDonell, S.G. "Fuzzy Logic Techniques for Software Metric Models of Development Effort". W. Pedrycz and J.F. Peters (eds) *Computational Intelligence in Software Engineering*. World Scientific: Singapore, 1998, 321-338

[8] Albrecht, A.J. and Gaffney, J.R. Software Function, Source Lines of Code, and Development Effort Prediction: a Software Science Validation, *IEEE Trans Soft Eng* 9(6), 1983, 639-648

[9] Boehm, B.W. *Software Engineering Economics*, Prentice-Hall: Englewood Cliffs, N.J., 1981

[10] Host, M. and Wohlin, C. "A Subjective Effort Estimation Experiment", *Info Softw Tech* 39, 1997, 755-762

[11] Heemstra, F. J. and Kusters, R.J. "Function Point Analysis: Evaluation of a Software Cost Estimation Model". *European Jnl Info Systems* 1(4), 1991, 229-237

[12] Hughes, R.T. "Expert Judgement as an Estimating Method", *Info Softw Tech* 38, 1996, 67-75

[13] Ohlsson, M.C., Wohlin, C. and Regnell, B. "A Project Effort Estimation Study", *Info Softw Tech* 40, 1998, 831-839

[14] Yager, R.R. and Filev, D.P. "Essentials of Fuzzy Modeling and Control". Wiley: New York, 1994

[15] Dubois, D. and Prade, H. *Fuzzy Sets and Systems: Theory and Applications*. Academic Press: London, 1980

[16] Miyazaki, Y., Terakado, M., Ozaki, K. and Nozaki, H. "Robust Regression for Developing Software Estimation Models", *Journal of Systems and Software* 27, 1994, 3-16

[17] Kitchenham, B. and Linkman, S. "Estimates, Uncertainty, and Risk", *IEEE Software* May/June 1997, 69-74

[18] Putnam, L.H. and Myers, W. "How Solved is the Cost Estimation Problem?", *IEEE Software* Nov/Dec 1997, 105-107

[19] Gray, A.R. and MacDonell, S.G. "Fuzzy Logic for Software Metric Models Throughout the Development Life-cycle", In *Proc 1999 Annual Meeting of the North American Fuzzy Information Processing Society - NAFIPS.* IEEE Computer Society Press, 1999, 258-262

[20] MacDonell, S.G., Gray, A.R. and Calvert, J.M. "FULSOME: A Fuzzy Logic Modeling Tool for Software Metricians", In *Proc 1999 Annual Meeting of the North American Fuzzy Information Processing Society - NAFIPS.* IEEE Computer Society Press, 1999, 263-267

[21] Bastani, F.B., DiMarco, G. and Pasquini, A. "Experimental Evaluation of a Fuzzy-set Based Measure of Software Correctness Using Program Mutation", In *Proc 15th Intl Conference on Software Engineering*, IEEE Computer Society Press, Los Alamitos CA., 1993, 45-54

[22] Kumar, S., Krishna, B.A. and Satsangi, P.S. "Fuzzy Systems and Neural Networks in Software Engineering Project Management". *Journal of Applied Intelligence* 4, 1994, 31-52

[23] de Ru, W.G. and Eloff, J.H.P. "Risk Analysis Modelling with the Use of Fuzzy Logic", *Computers & Security* 15 (3), 1996, 239-248

[24] Khoshgoftaar,. T.M., Evett, M.P., Allen, E.B. and Chien, P.D. "An Application of Genetic Programming to Software Quality Prediction", W. Pedrycz and J.F. Peters (eds) *Computational Intelligence in Software Engineering.* World Scientific: Singapore, 1998, 175-196

[25] Khoshgoftaar, T.M., Allen, E.B., Jones, W.D. and Hudepohl, J.P. "Which Software Modules Have Faults Which Will Be Discovered by Customers?" *Journal of Software Maintenance: Research and Practice* 11, 1999, 1-18

[26] Ebert, C. "Experiences with Criticality Predictions in Software Development", ACM SIGSoft SEN 22(6), 1997, 278-293

[27] Wiegers, K.E. *"Read my lips: no new models", IEEE Software* Sept/Oct 1998, 10-13

[28] MacDonell, S.G., Shepperd, M.J. and Sallis, P.J. "Metrics for Database Systems: an Empirical Study", In *Proceedings of the Fourth International Symposium on Software Metrics (Metrics '97)*, Albuquerque NM, IEEE Computer Society Press, 1997, 99-107

[29] Shepperd, M.J. and Kadoda, G. "Using Simulation to Evaluate Prediction Techniques", In *Proceedings of the 7th Intl Symposium on Software Metrics, London*, IEEE Computer Society Press, 2001, 349-359

[30] Pickard, L., Kitchenham, B.A. and Linkman, S. "An Investigation of Analysis Techniques for Software Datasets", In *Proceedings of the Sixth International Symposium on Software Metrics (Metrics '99).* Boca Raton FL, USA, IEEE Computer Society Press, 1999, 130-142

[31] Miyazaki, Y., Terakado, M., Ozaki, K. and Nozaki, H. "Robust Regression for Developing Software Estimation Models", *Journal of Systems and Software* 27, 1994, 3-16

Integrating Genetic Algorithms With Systems Dynamics To Optimize Quality Assurance Effort Allocation

Balasubramaniam Ramesh[1] and Tarek K. Abdel-Hamid[2]

[1]Department of Computer Information Systems
Georgia State University
Atlanta, GA 30303
bramesh@gsu.edu

[2]Naval Postgraduate School
Monterey, California 93943
tkabdelh@nps.naby.mil

ABSTRACT

Optimal allocation of effort towards Software Quality Assurance is critical for the successful development of software systems. A decision support system based on systems dynamics simulation model has been developed to support software project managers in making this decision. The effectiveness of the decision support system has been enhanced with a genetic algorithm to optimize this decision variable. The architecture of the DSS, and results from an empirical experiment to validate the effectiveness of our approach are presented.

KEYWORDS

Software quality assurance, Project Management Decision Support, Genetic Algorithms, Systems Dynamics.

1. INTRODUCTION

Software quality assurance (QA) constitutes a set of activities undertaken during the development of software systems so as to reduce the risks of unacceptable system performance and to ensure that the produced software does conform to established technical requirements [Pressman 2001]. Increasingly, software quality assurance is being recognized as a critical factor in the successful development of software systems. The reason is a simple one: the evidence indicates that failure to pay attention to QA often results in unsatisfied users and higher lifecycle costs. Software quality assurance is approached through two complementing strategies. First, through ensuring that the quality is initially built into the product. This involves emphasizing the early generation of a coherent, complete, unambiguous, and non-conflicting set of user requirements. Then, as the product moves into the design and coding stages, a second set of QA tools are deployed to continuously review and test the system (e.g., through walkthroughs, inspections, code reading, etc.) [Pressman 2001]. Even in the context of systems that are developed at Internet speed the ability to understand and control the level of quality is important to achieve the desired balance between quality and speed [Baskerville *et al.* 2001].

The utilization of QA tools and techniques does, however, add significantly to the cost of developing software as person-days are expended in developing test cases, running test cases, conducting structured walkthroughs, etc. Consider, as an example, the Omega project, in which 180 development staff worked to port a micro-networking operating system to new hardware [Gilb 1988]: Initial planning estimates indicated 30,000 person hours a year for inspections. Management assumed that a third of this resource would be spent on inspection meetings, and the average meeting would last 90 minutes and involve five persons. This meant that there would be well over 1300 inspection meetings, requiring 1300 conference room bookings. As there were no conference rooms in the office area, two were quickly built!

It is no wonder then, that the added cost of QA activities is a source of concern to QA managers, program managers and customers. As of yet this concern has not been adequately addressed in the literature. Our objective in this article is to present a tool to support the software project manager in efficiently allocating the QA effort throughout the software development lifecycle. This tool uses a systems dynamics simulation model that provides the ability for project managers to experiment with decision variables such as software quality assurance effort allocation. A genetic algorithm has been integrated with this model to find an optimal QA effort allocation scheme.

2. A CASE STUDY

Consider the case of a software project conducted to develop a software system for a space application. The system was estimated to be: 16,000 delivered source instructions (DSI) in size; require 1,100 person-days for development and testing; and be completed in 320 working days. How much should management allocate to QA?

The above project is not a hypothetical scenario. Indeed, it is a real project that was conducted at one of NASA's space flight centers. The basic requirements for the project were to design, implement, and test a software system for processing telemetry data and for providing altitude determination and control for a NASA satellite.

As is typically the case, resources for QA were allocated as a function of the project's total development effort. In this case, approximately 30% of the project's development resources were allocated to QA, a level that is significantly higher than the industry norm [Dunn 1994]. Upon completion, the project's statistics were:

Project size	24,000 DSI
Development cost	2,200 person-days
Completion time	380 working days

Obviously the project was not a total success. On the positive side, the end product was reported to be of high quality i.e., was reliable, stable, and easy to maintain. On the other hand, the project overshot its schedule by 20% and its cost by 100%.

In a project postmortem, a number of issues were raised including the issue of whether the QA effort allocated was optimal. And if not, what the impact was on the project's cost and schedule. In principle such issues can be addressed by conducting a controlled experiment in which the project is repeated many times under varied QA expenditure levels. Such an experimental approach, however, is obviously too costly and time consuming to be practical. Furthermore, even when affordable, the isolation of the effect (cost) and the evaluation of the impact of any given practice (QA) within a large, complex, and dynamic social system such a software project environment can be exceedingly difficult.

Simulation modeling, on the other hand, does provide a viable alternative for such a task. In addition to permitting less costly and less time-consuming experimentation, simulation-type models make "perfectly" controlled experimentation possible. Indeed:

> The effects of different assumptions and environmental factors can be tested. In the model system, unlike the real system, the effect of changing one factor can be observed while all other factors are held unchanged. Such experimentation will yield new insights into the characteristics of the system that the model represents. By using a model of a complex system, more can be learned about internal interactions than would ever be possible through manipulation of the real system [Forrester 1961].

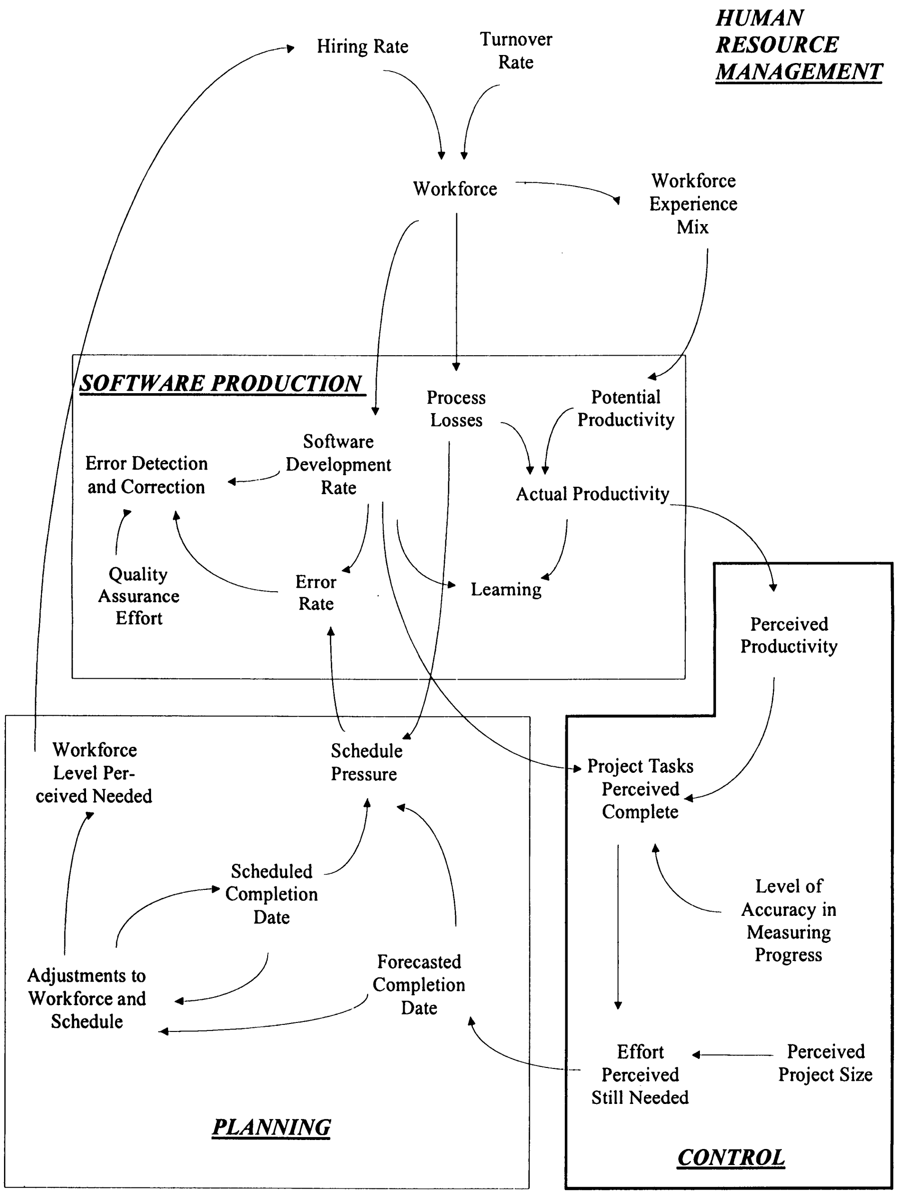

HUMAN RESOURCE MANAGEMENT
Hiring Rate
Turnover Rate
Workforce
Workforce Experience Mix
SOFTWARE PRODUCTION
Process Losses
Potential Productivity
Software Development Rate
Error Detection and Correction
Actual Productivity
Quality Assurance Effort
Error Rate
Learning
Perceived Productivity
Workforce Level Perceived Needed
Schedule Pressure
Project Tasks Perceived Complete
Scheduled Completion Date
Level of Accuracy in Measuring Progress
Adjustments to Workforce and Schedule
Forecasted Completion Date
Effort Perceived Still Needed
Perceived Project Size
PLANNING
CONTROL

Figure 1. Four Subsystems of the System Dynamics Simulator.

3. A SYSTEM DYNAMICS MODEL OF SOFTWARE DEVELOPMENT

As part of a wide-ranging study of software project management, a comprehensive system dynamics model of the software development process has been developed. The model was developed on the basis of field interviews of software project managers in five organizations, complemented by an extensive database of empirical findings from the literature. The model integrates the multiple functions of the software development process, including both the management-type functions (e.g., planning, controlling, and staffing) as well as the software production-type activities (e.g., designing, coding, reviewing, and testing). Figure 1 is an overview of the simulation model's four major subsystems: (1) the human resource management subsystem; (2) the software production subsystem; (3) the controlling subsystem; and (4) the planning subsystem.

As the model is quite comprehensive and highly detailed, it is infeasible to fully explain it in the limited space of this article. Therefore, the description will be limited to a high level overview of the four subsystems. The model's QA component (which is part of the software production subsystem), is, however, discussed in more detail in the following section. (For a more detailed description of the model's structure, its mathematical formulation, and its validation the interested reader can refer to [Abdel-Hamid and Madnick 1991].

The **human resource management** subsystem captures the hiring, training, and transfer of the human resource. Such actions are not carried out in a vacuum, but are affected by the other subsystems; for example, the hiring rate is a function of the work force level needed to complete the project by a given date. Similarly, the available work force has a direct bearing on the allocation of manpower among the different production activities.

As the software is developed, it is also reviewed using quality assurance activities such as structured walkthroughs to detect any errors. Errors detected through such activities are reworked. However, some errors "escape" detection until the testing phase. The development lifecycle phases incorporated in the **software production subsystem** include the designing, coding, and testing phases. Three sets of factors affect the error generation rate. The first set includes organizational factors (e.g., the organization's use of structured techniques, the overall quality of the staff). A second set includes project-specific factors (e.g., project complexity, system size, programming language). While these two sets of factors differ from organization to organization and from one project to another, they tend to remain constant throughout the development lifecycle of any single project. A third set of factors affecting error generation includes the work force mix and schedule pressures. Unlike the factors in the first two sets, these two variables change dynamically throughout the lifecycle.

As progress is made, it is reported. A comparison of the degree of project progress to the planned schedule is captured within the **control subsystem**. Once an assessment of the project's status is made, it becomes an important input to the planning function. In the **planning subsystem**, initial project estimates are made and then revised, when necessary, throughout the project's life. For example, to handle a project that is behind schedule, plans can be revised to (among other things) hire more people, extend the schedule, or do both.

4. SOFTWARE QUALITY ASSURANCE MODEL

A primary objective of the software quality assurance (QA) activity is to detect the software errors that have been generated. Typically, the QA effort is planned and allocated as a fixed schedule of periodic group-type functions e.g., as two-hour walkthroughs scheduled once a week [Abdel-Hamid and Madnick 1991]. During these periodic "QA windows," all tasks developed since the previous review are supposed to be processed. A surprising finding showed that all completed tasks, irrespective of how many these were, were always indeed "processed." No backlogs, therefore, develop in the QA pipeline even when QA activities are suspended temporarily because of schedule pressures. For example, when walkthroughs are suspended on a project, the requirement to review the affected tasks is bypassed, not postponed. Since the objective of the QA activity is to detect errors and since undetected errors are by their very nature invisible, it is almost impossible to tell whether an adequate QA job was done (except much later in the lifecycle). Under such circumstances it is easy to rationalize both to oneself and to management that the QA job that was "convenient" to do, was not insufficient. Furthermore, the QA effort that is convenient to expend (given scheduling considerations) is usually never exceeded even when more effort is called for. There seems to be no significant incentives to do otherwise. First, at a psychological level, there are actually disincentives for working harder at QA, since it only "exposes" more of one's mistakes. Second, at the organizational level, there are seldom any real reward mechanisms in place to promote quality or quality-related activities [Abdel-Hamid and Madnick 1991].

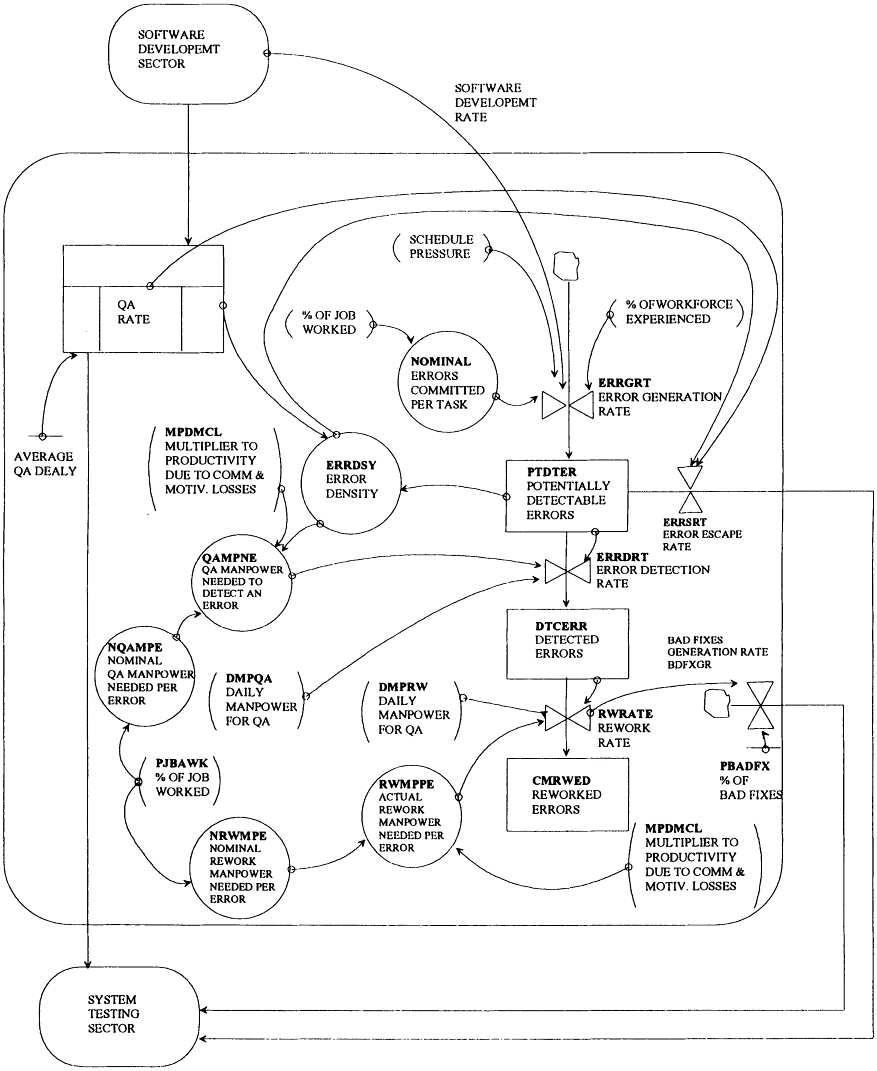

SOFTWARE DEVELOPEMT SECTOR
SOFTWARE DEVELOPEMT RATE
QA RATE
SCHEDULE PRESSURE
% OF JOB WORKED
% OFWORKFORCE EXPERIENCED
NOMINAL ERRORS COMMITTED PER TASK
ERRGRT ERROR GENERATION RATE
AVERAGE QA DEALY
MPDMCL MULTIPLIER TO PRODUCTIVITY DUE TO COMM & MOTIV. LOSSES
ERRDSY ERROR DENSITY
PTDTER POTENTIALLY DETECTABLE ERRORS
ERRSRT ERROR ESCAPE RATE
QAMPNE QA MANPOWER NEEDED TO DETECT AN ERROR
ERRDRT ERROR DETECTION RATE
DTCERR DETECTED ERRORS
BAD FIXES GENERATION RATE BDFXGR
NQAMPE NOMINAL QA MANPOWER NEEDED PER ERROR
DMPQA DAILY MANPOWER FOR QA
DMPRW DAILY MANPOWER FOR QA
RWRATE REWORK RATE
PJBAWK % OF JOB WORKED
CMRWED REWORKED ERRORS
PBADFX % OF BAD FIXES
RWMPPE ACTUAL REWORK MANPOWER NEEDED PER ERROR
NRWMPE NOMINAL REWORK MANPOWER NEEDED PER ERROR
MPDMCL MULTIPLIER TO PRODUCTIVITY DUE TO COMM & MOTIV. LOSSES
SYSTEM TESTING SECTOR

Figure 2. Model Structure for Error Detection/Correction.

Figure 2 depicts the system dynamics model's structure for the detection and correction of errors. This component of the model together with two others – software development and system testing – constitute the software production subsystem of Figure 1. To capture this "Parkinsonian" nature of the QA activity, the QA RATE shown in Figure 2 is modeled as an exponential delay. This says that software tasks that are developed will always be QA'ed (or, more accurately, considered QA'ed) after a certain delay, which is independent of the actual QA effort allocated!

4.1 Error Detection Factors

However, the effectiveness of QA would obviously depend on the QA effort. That is, the amount of errors that are detected will necessarily be a function of the amount of QA effort allocated. This is evident in Figure 2, where the error DETECTION RATE equals DAILY MANPOWER FOR QA divided by QA MANPOWER NEEDED TO DETECT AN ERROR.

The QA MANPOWER NEEDED TO DETECT AN ERROR, is a function of both error-type (e.g., design errors versus coding errors) and on the efficiency of how people work. Work inefficiencies such as man-hours lost on communication and other non-project activities (e.g., personal business, coffee breaks, etc.) are captured by the model's MULTIPLIER TO PRODUCTIVITY DUE TO COMMUNICATION AND MOTIVATION LOSSES.

Finally, there is the effect of error density on the error detection activity. At any point in time, the set of POTENTIALLY DETECTABLE ERRORS constitutes a hierarchy of errors, in which some are subtler, and therefore more expensive to detect than others. Empirical results reported by Basili and Weiss [1985] suggest that the distribution is pyramid-like, with the majority of errors requiring a few hours to detect, a few errors requiring approximately a day to detect, and still fewer errors requiring more than a day to detect. In the model it is assumed that as QA activities are performed, the more obvious errors will be detected first. As these are detected, it becomes increasingly expensive to uncover the remaining, more elusive (although less pervasive) errors.

4.2 Error Correction Factors

Errors detected through QA are reworked. The REWORK RATE is a function of how much effort is allocated to the rework activity (DAILY MANPOWER FOR REWORK) and the ACTUAL REWORK MANPOWER NEEDED PER ERROR. The ACTUAL REWORK MANPOWER NEEDED PER ERROR has two components. The first is the NOMINAL REWORK MANPOWER NEEDED PER ERROR, which is a function of error-type, i.e., design versus coding errors. Design-type errors are thus generated at a higher rate, are more costly to detect, and are more costly to rework [Pressman 2001]. The ACTUAL REWORK MAN-POWER NEEDED TO CORRECT AN ERROR also depends on the work efficiency of the project staff. That is, the communication and motivation losses must be accounted for. For example, if the MULTIPLIER TO PRODUCTIVITY DUE TO COMMUNICATION AND MOTIVATION LOSSES is 0.5 (indicating that 50 percent of the time is lost in nonproductive activities), then the actual rework manpower needed to correct an error becomes twice the nominal.

As further demonstrated in Figure 2, the reworking of software errors is not, itself, an errorless activity. As detected errors are reworked, some fraction of the corrections will be bad-fixes. The detection and correction of such bad-fixes, together with errors that escape QA detection during the project's development phases, are activities that are captured in the model's system testing sector.

5. A DECISION SUPPORT SYSTEM (DSS) FOR QUALITY ASSURANCE EFFORT ALLOCATION

Controlled experimentations are very costly and time consuming in software engineering [Myers 1976]. Even when experimentation is affordable, the isolation of the effect of any given practice within a large, complex and dynamic project environment can be extremely difficult [Glass 1982]. Simulation modeling offers a viable alternative for testing software engineering hypotheses. The Systems Dynamics Simulation has been used to support decision-making in software project management in a variety of contexts. For example, it has been used to support a continuous resource planning and allocation activity. Software managers use the model to update project cost and schedule estimates due to factors such as changes in user requirements [Abdel-Hamid 1993]. The model also allows managers to conduct trade-off analyses, e.g., between extending the duration of a project versus adding more personnel. In this research, the Systems Dynamics Simulation is used to study the allocation of effort towards software quality assurance.

The QA effort is a function of a large number of factors that are interrelated in a complex non-linear fashion. For example, one of the critical factors, cost of undetected errors, grows exponentially and not linearly over time. Therefore, analytical solutions to develop an "optimal" QA scheme are not available. The systems dynamics simulator provides a viable tool to assess QA effort allocation scenarios. In the simulation model, managerial policies, such as the QA effort allocation throughout the life cycle, are captured as table functions. This representation allows a policy variable (e.g., person-days allocated to QA over time) to be defined as a function of project life cycle stage or the project completion status. Once a policy is defined, the model can be run to assess its impact on project performance. Such experimentation can help a project manager make informed decisions.

Specifically, simulation can be run with any QA effort allocation scheme (keeping all the other parameters constant). At the end of the simulation run, the total project cost is estimated by the simulation system. The user can vary the QA effort allocation scheme and study its effect on total project cost. By experimenting with various schemes, the user may decide on an appropriate scheme.

However, this process has a serious limitation. The QA effort allocation as a fraction of total effort can vary theoretically between 0% and 100% for any period. If the life cycle is divided into 10 periods (10% complete, 20% complete etc.) and the decision maker has to select a specific value in this range for each of the 10 periods, then the total number of possible QA schemes will be 100^{10}. The decision maker can, however, use his domain knowledge to limit the search space. For example, it can be decided that very high (say, over 75%) or very low (say, below 10%) levels of QA effort are undesirable from implementation and productivity considerations. Even with such constraints, the search space for potential solutions is extremely large. Running the simulation manually to find an optimal solution can, therefore, be very difficult. The development of an automated procedure to try different values of the decision variables to find the "optimal" solution can be significantly enhance the usefulness of the systems dynamic simulator as a DSS.

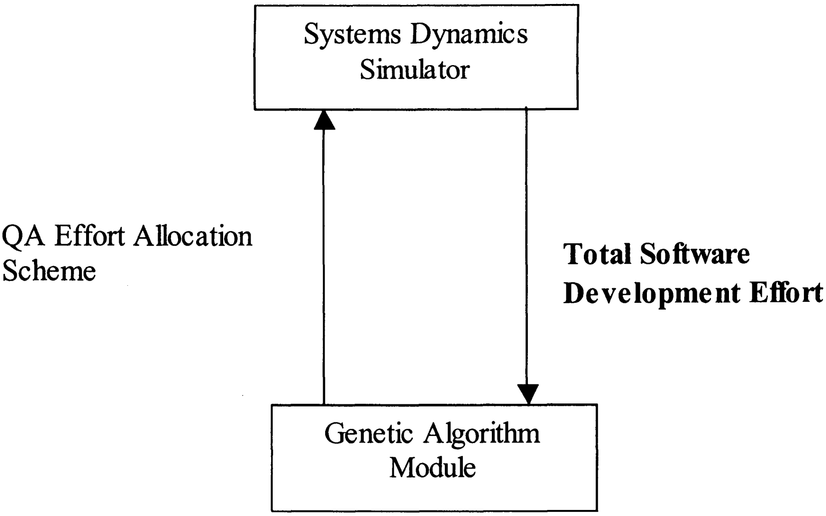

Systems Dynamics Simulator
QA Effort Allocation Scheme
Total Software Development Effort
Genetic Algorithm Module

Figure 3. Overview of the System Architecture.

5.1 Genetic Algorithm To Optimize QA Effort Allocation

The search for a "good" solution can be done using classical exhaustive search methods if the search space is small. However, when the search space is very large, such as in the case of the QA allocation problem, computational intelligence techniques such as Genetic Algorithms (GA) have been found to be very effective. GAs are very effective in searching complex, non-linear, multidimensional search spaces even in the absence of specific knowledge about the problem domain. GAs are probabilistic algorithms that combine features of stochastic and directed search and are more robust than most existing search methods [Michalewicz 1996]. The problem of finding an optimal QA allocation scheme can be thought of as a dynamic non-linear optimization problem in which the objective is to minimize the total development effort. GAs have been used successfully in optimizing system behavior in such problems [Sholtes 1994].

Figure 3 provides the architecture of a DSS in which the Systems Dynamics simulator is coupled with a Genetic Algorithm module for optimizing the model behavior. The GA module proposes different QA effort allocation schemes. These are provided as inputs to the simulation model. The simulation model is executed and estimates the total software development effort with that scheme. This information is used by the GA to develop an efficient QA scheme.

5.2 Genetic Algorithms

Genetic Algorithms are a class of search, adaptation and optimization techniques that are based on the principles of natural evolution in which individuals and species that adapt to changing environments have a higher chance of survival [Forrest 1993; Srinivas and Patnaik 1994; Forrest 1996]. The features that distinguish individuals are determined by a set of genes or chromosomes. Selection involves the principle of "survival of the fittest" that characterizes natural evolution. It implies that the fittest genes survive during evolution. Reproduction involves the combination of the genetic material of parents to form the genetic material of offsprings. Crossover refers to the exchange of segments of chromosomes of the parents during reproduction. Mutation involves random changes in the genetic make up. Genetic algorithms attempt to use these principles of crossover, mutation and recombination and a variety of mechanisms inspired by natural evolution to solve search or optimization problems.

In a GA, solutions to a problem are represented as (often binary) strings. A one-to-one mapping between the strings and the actual solution to the problem must exist. A fitness function that measures the quality of the solution is either a formal objective function (e.g., directly computed or results from a simulation) or a subjective judgment. The basic steps in finding an "optimal" solution are:

1. Create a population of initial solutions randomly
2. Evaluate the fitness of each individual in the population
3. Create a new population using genetic operators (such as selection, crossover and mutation)
4. Repeat steps 2 & 3 until the termination condition (such as convergence, resource limitation) is reached.
5. Select the individual with the best fitness value as the solution to the problem in that generation.

This process mimics evolution in achieving novelty in its approaches to maintaining fitness [Levy 1992]. Genetic algorithms generate high quality solutions but have fewer tendencies to terminate on local optima than traditional techniques. Genetic algorithms outperform traditional learning techniques, especially when the solutions that have to be learned are complex. They are especially useful when there is no domain knowledge available to guide the search for solutions [Holsheimer and Siebes 1994] or when the noisy data is used in data mining [Goldberg 1994].

5.2.1 How do GAs work

The notions of schemata and building blocks proposed by Holland [Holland 1992] are used in explaining the working of a GA. In simple terms, schemata are similarity templates or subsets that have some features in common. Building blocks are schemata that have high fitness values. These are preferred in selection and exchanged by genetic operations. The building-block hypothesis states that the strings with high fitness values can be identified by sampling and combining building blocks. The fundamental theorem governing the working of a GA is Holland's schemata theorem, which states that the schemata with high fitness values grow exponentially with time. Later empirical and theoretical research [Goldberg *et al.* 1992; Koza *et al.* 1996] provides more insights into the workings of a GA.

5.2.2 Comparison with other techniques

Genetic algorithms differ considerably from other search and optimization techniques such as hill-climbing and random walk in several ways [Goldberg 1989]:

- The parameters of the functions to be optimized are coded in a GA, typically as fixed length strings.
- GA employs a highly parallel search using a population of potential solutions, rather than a single search point.
- The fitness of a solution is determined by a payoff or cost function, rather than auxiliary information (such as derivatives used in gradient techniques or the various tabular parameters needed by combinatorial optimization)
- GAs employ probabilistic rather than deterministic rules to guide the search towards regions in the search space that are likely to improve performance.

Genetic algorithms are increasingly popular to solve problems in a variety of domains including engineering, economics, management science and other areas (see [De-Jong 1999] [Holland 1992] for surveys). Building on the principles of GA, Koza introduced genetic programming (GP) [Koza 1992]. GP uses symbolic expressions (S-expressions)- rather than bit strings - as units being evolved by a genetic program. A Genetic programming has been successfully used in automatic induction in a wide variety of applications ranging from generating small subroutines to real-time problems such as robot control. Koza [Koza 1994; Koza *et al.* 1999; Koza 2000] asserts that GP has already achieved the goal of producing results that equal or exceed human performance in a variety of domains such as algorithm design, game playing, pattern recognition, control and design. Taking a high-level statement of a problem's requirements, GP is capable of producing solutions that infringe on or improve on previously issued patents in problems like circuit design [Koza *et al.* 1999].

The application of using GP to solving business problems is exemplified by a system developed by Dworman, Kimbrough and Laing [Dworman *et al.* 1995] that discovers high quality negotiation patterns in a multi-agent game. Though this approach has similarities to our work on integrating a systems dynamics simulation with GAs, our approach has the benefit of learning from data drawn from a validated systems dynamics model. In contrast, the outcomes of the negotiations in the multi-agent game can be very easily evaluated for optimum results.

Holland [Holland 1992] establishes the appropriateness of using Genetic Algorithms for learning patterns such as the QA effort allocation scheme. The GLOWER system [Dhar *et al.* 2000] represents an approach to learning rules from data using genetic algorithms. The system exploits the power of GA to scour the search space thoroughly in finding interesting trading rules. This approach is especially useful when the search space includes continuous variables. Unlike greedy search processes used by machine leaning systems, Genetic Algorithms are less constrained in searching for all applicable rules. In GLOWER, the chromosomes are made up of sets of genes that represent constraints on a single descriptor variable. The characteristics of our search space are similar to those used in GLOWER. However, instead of relying on data from past trades, our system learns from data drawn from the systems dynamics simulator.

Simulated annealing, genetic algorithms and evolutionary strategies are similar in that they use probabilistic search mechanisms to find the best solution. Simulated annealing generates a sequence of states that converge to an optimum solution based on cooling schedules. The principal difference between evolutionary strategies and genetic algorithms is that the former uses mutations as the primary search mechanism, whereas GA uses crossover and mutation as search mechanisms.

6. THE NASA SOFTWARE PROJECT REVISITED

In this section, we describe how a GA can be used to find an optimal QA resource allocation scheme for the NASA software project.

6.1 Representation

The representation of the problem in terms of strings that can be manipulated by the GA is a critical step in successful application of the method. Our problem involves finding the "best" QA effort allocation scheme, specified as a vector representing the fraction of the total resources employed in QA during each period.

6.1.1 Limiting the search space

Knowledge about the problem domain can be used to limit the search space of potential solutions so that the GA can converge to an "optimal" solution quickly. Theoretically, the values of QA allocation in each period can range from 0% (no QA at all) to 100% (all resources allocated to QA) during each development phase. However, it is unreasonable to assume extremely high levels of QA effort in large scale projects. The return on the QA investment typically flattens out when it exceeds the 40% of the development effort [Abdel-Hamid 1988]. Similarly, using very low levels of QA allocation may not be feasible from a project manager's perspective; a project manager may want to maintain at least a minimal QA effort during all phases of the life cycle for reasons of gradual staffing, training, and maintaining a QA "presence". Under these conditions, the range of candidate values for the GA to select may be set between say, 10% and 75% (to allow sufficient room for plausible values)[1].

Though the GA may find the same optimal solutions even with the larger search spaces, it is more efficient to eliminate infeasible regions to speed up convergence towards the solution. However, care must be exercised in constraining the search space to minimize the risk of eliminating potentially optimal solutions from consideration.

6.1.2 Fitness measure

The total development effort for a project is the sum of the effort required in the design, coding, QA, rework and testing. As discussed in Section 3, the QA activity has significant effect on these activities. Therefore, the total development effort is an appropriate measure of fitness of a QA effort allocation scheme. It may be used as a surrogate for the total project cost that should be minimized. When provided with a candidate QA effort allocation scheme, the simulation model can compute the total development effort for the project that is used as the measure of fitness of the scheme.

Specifically, the GA generates candidate QA effort allocation schemes. Each scheme is passed onto the simulation for evaluation. The simulation is executed with the scheme as the input (keeping all the other parameters constant). The simulation estimates the total project cost and returns it to the GA. The GA uses this total project cost as the fitness value of the scheme. The objective of the GA is to evolve a QA effort allocation scheme that minimizes this cost.

6.2 GA parameters

A variety of parameters can be controlled while evolving solutions using a GA. The population size, total number of generations, frequency of crossover and mutation are some of the important variables. The choice of the 'best' values for each of these variables is typically determined by varying each of the parameters over a range of values and observing the effect on convergence of a solution. Several "rules of thumb" for different classes of problems are also discussed in the literature [Koza 1992; Koza *et al.* 1996]. Often, simple hand optimization is used by starting with 'standard' parameter settings and changing each parameter one at a time and see what results are obtained. In our experiments, crossover rates between 0.6 and 0.9 were used to examine the effect of various values. Mutation rates were varied between 0.01 and 0.15 and the population sizes ranging from 500 to 1500 were used in the experiments. The values of the various parameters that produced the best results is shown in Table 1. It should be noted that such parameter tuning by experimentation may lead to sub-optimal choice of these parameters. A variety of techniques for find good parameter settings for genetic algorithms have been discussed in the literature [Grefenstette 1986]. However, the problem of controlling parameters of an evolutionary algorithm is still a subject of active research and theoretical investigations on selecting optimal parameters do not provide results with wide generalizability {Eiben, 2000 #19. Further, the cost of finding such parameter tuning could be very significant and the rewards in solution quality may not justify the cost [Beasley *et al.* 1993].

The experiment uses a crossover-dominated GAs (i.e., with a low mutation rate). Here, mutation is performed randomly on a gene of a chromosome and it ensures that every region of the problem space can be reached. When a gene is mutated it is randomly selected and randomly replaced with another symbol from the alphabet. To eliminate the risk of pathological initial populations in which an important low-order schema my be missing, and needs to be created by only by mutation, the GA creates a super-uniform initial population in which all schemata are equally represented using a reduced-variance stochastic algorithm which produces a population with no local, but large global correlations [Schraudolph and Belew 1992; Schraudolph and Grefenstette 1992]. The trials are stopped when the maximum generation is reached or the convergence threshold (the percentage of the population that needs to have the same values for it to be considered to have converge) is achieved. A high convergence threshold implies that the genetic make up of the population is not significantly different enough to produce better results with more trials.

Parameter	Value
Population Size	1000
Max. Generations	100
Crossover Rate	0.60
Mutation Rate	0.05
Convergence Threshold	0.95

Table 1. GA parameters.

6.3 Incorporating constraints

Problem specific constraints can be easily incorporated to guide the GA towards desirable solutions and away from those that are undesirable. This can be done by incorporating in the fitness function, a penalty for solutions that have desirable characteristics. Similarly, a reward for solutions that have desirable characteristics can be incorporated into the fitness function. For instance, the management may specify that the QA effort allocation may not vary by more than say 5% between periods, except in the initial stages of the project. Such a constraint can be easily incorporated in a GA using a penalty function as a part of the fitness function. Solutions that violate the following constraint may be assigned a very high fitness value (in this fitness minimization problem) so that they will become unattractive:

$$ABS[QA_t\text{-}QA_{t+1}] > 5\%$$

When such a fitness consideration is used, a scheme that provides a smooth QA scheme will perform well. Thus, through the process of careful evaluation of potential solutions, a project manager may identify and impose constraints on the solution space. Finding the best possible solution within those constraints is well achieved by the GA.

6.4 Results from GA

The GA produced several hundred schemes that did better than the actual NASA project in terms of the total project cost. Figure 4 shows the best QA effort allocation scheme suggested by the GA. This scheme would require only 1475.9 person-days for the project compared to the 2200 person-days required in the actual project. In other words, by adopting the QA effort allocation scheme suggested by the GA, the project could have saved nearly a third of the total project costs.

7. SIMULATION EXPERIMENTS

Though the above results from the GA are impressive, it is useful to evaluate its performance against other possible approaches. However, as discussed earlier, no analytical formulations to this problem are available. Therefore, it is not possible to use traditional analytical procedures to find a solution. However, the results from the GA can be compared to those developed by other procedures. We have chose two candidates:

- a series of randomly generated QA effort allocation schemes. and
- QA effort allocation schemes used by human decision makers using the systems dynamics simulator as a DSS.

The performance of the GA can be compared to the results from these experiments on the basis of both

- the total development effort as a measure of the impact of the solution on total project costs, and
- qualitative assessment of the implementability of the proposed schemes.

This section describes the simulation experiments that were conducted for such a comparison.

7.1 Randomly Generated Schemes

The first of the simulation experiments involved randomly creating several QA effort allocation schemes and running to simulation experiment to evaluate the effectiveness of these schemes on the basis of the total development effort produced by the simulation. These experiments were based on the premise that given access to a simulator, a decision maker will be able to experiment with a large number of possible QA effort allocation schemes and may be able to find an appropriate scheme.

In the experiment 2000 QA effort allocation schemes were randomly created. Each was passed on to the simulation for the estimation of the total development effort. Table 2 summarizes the results from these runs.

Random QA Scheme	Total Development Effort (Person-Days)
"Best" scheme	1541.38
"Worst" scheme	14700.70
Average	3321.86
Standard Deviation	2405.09
Actual project	2200.00

Table 2. Results from Random Schemes.

During this experiment, the average total effort from randomly created schemes was well above the level experienced in the project. The worst scheme increased the development effort more than six fold. However, the best scheme generated a QA scheme that was about 30% lower than that observed in the actual project. But this scheme did not perform as well as the best scheme developed by the GA. As the GA starts with a set of random schemes and evolves better performing solutions based on feedback from the simulation, this result is not surprising. It should be noted that even with such a simulation available, human decision makers are unlikely to be able to evaluate thousands of schemes manually to find the best scheme. Instead, the use of a GA to automate and optimize this task appears a more reasonable approach.

7.2 Manual Simulations

Another experiment was set up in which subjects played the roles of project managers making QA resource allocation decisions over the life of a software project. As project managers, the subjects were required to use the what-if capabilities of the model to derive "optimal" QA allocation for the project. The objective function was to minimize the total project cost as measured by the total effort. The simulation was based on the NASA project and the QA scheme was the only variable manipulated by the subjects.

The experiment was conducted with 25 graduate students at a U.S. university. The subjects were master's students in a computer systems management curriculum, and had an average of 12 years of full time work experience. The experiment was part of a course on software engineering. The experiment was conducted on desktop computers with interactive simulation software system written in Dynamo. All students were given a hands-on demonstration on the use of the simulation package before the experiment. The experimental set up had been tested in a variety of contexts to ensure that the software performed as intended.

The subjects were also given a one-hour tutorial on software project management, explaining the key principles involved. Further, they had undergone graduate level work in the area of software engineering. Each subject was provided a five-page set of written instructions. A pilot was conducted to ensure that the subjects understood the instructions properly.

The experimental task involved the development of different QA effort allocation schemes which were used as inputs in simulation runs. At the end of each run, the simulator provided information on percentage of defects detected, rework cost, testing cost etc. in addition to assessing the impact on project cost. The subjects would use this information to revise their QA effort allocation schemes, and rerun the simulation (with the objective of minimizing the total development effort) until they are satisfied with the results. No limits were placed on how many times the simulation can be run by the subjects. Each subject reported the "optimal" QA scheme and the effort estimate at the conclusion of the experiment.

7.2.1 Manual Experiment Results

Table 3 summarizes the results from the experiments and the result from the actual NASA project. The "best" scheme (with minimum total project cost), the "worst" scheme (with the maximum total project cost) developed by the subjects as well as summary statistics are presented.

Manual QA Scheme	Total Development Effort (Person-Days)
"Best" scheme	1487.42
"Worst" scheme	1730.57
Average	1585.80
Standard Deviation	69.83
Actual project	2200.00

Table 3. Results from Manual Simulation.

Several factors may explain the superior performance of the subjects compared to the actual project. In the actual project, the project manager had to rely only on prior limited experience on similar projects in arriving at a QA scheme. Subjects in the experiment had the benefit of experimenting with a variety of QA schemes using the Systems Dynamics simulation and observing the effect on total project costs. The subjects had conducted an average of eight trials before arriving at their final solution. The results from the experiment indicate that using the simulation, a significantly improved solution may be found. Even the "worst" scheme from the experiment achieved a significantly lower cost compared to the actual project. The best scheme arrived at during the experiment would have saved more than 33% of the total project cost. The results illustrate the usefulness of the systems dynamics simulation as a DSS in making policy decisions such as QA effort allocation.

The examination of the QA schemes developed by the subjects in our simulation experiment suggests that they had used their knowledge of software engineering principles in limiting their search spaces. For instance, most subjects had developed QA schemes with high level of effort early in the life cycle, declining to a lower level later in the life cycle - consistent with practices advocated in software engineering literature. Similarly, no subject had used very high (over 70%) or very low (less than 10%) levels of QA effort indicating that they had considered values outside this range infeasible.

7.3 Comparison with GA results

A comparison of the results from the actual project, the best QA schemes from the GA, random generation and the manual experiment is presented in Table 4. Figure 4 graphically shows the corresponding QA schemes.

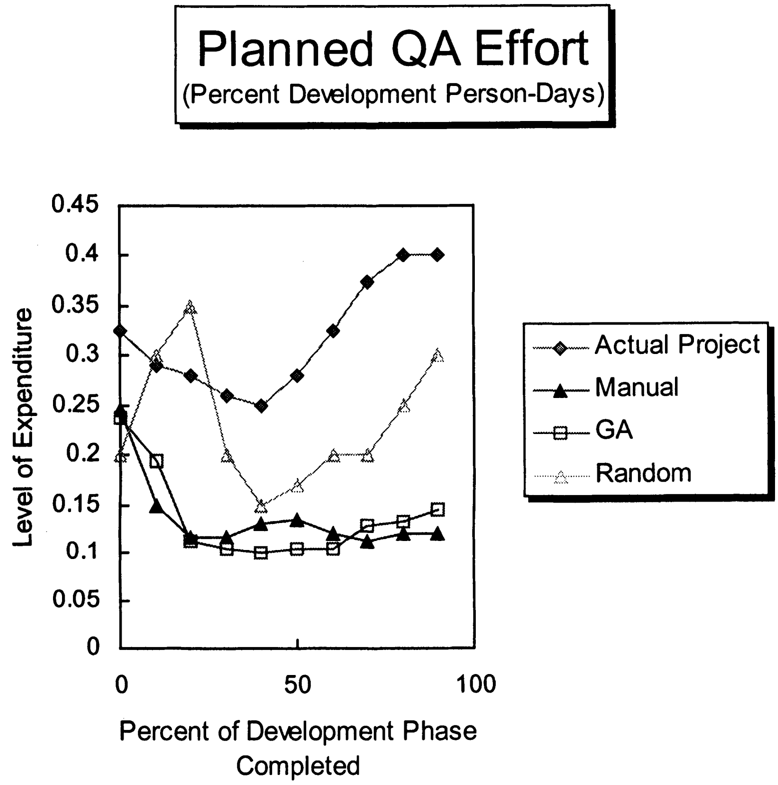


Figure 4. QA Effort Allocation Schemes.

7.3.1 Project Costs

The best scheme developed by the GA outperforms the manual simulation by about 12 person-days. In addition, the GA developed over 120 solutions that were better than the best solution produced manually. Also, the GA and the manual solutions outperformed the best results produced by the random scheme. The result is significant in that it the actual project costs were nearly 50% more than that would have been obtained by using any of these schemes suggested by the GA.

Source of QA Scheme	Total Development Effort (Person-Days)
Actual Project	2200
Random Scheme	1541.38
Manual	1487.4
GA	1475.9

Table 4. Project Cost under different QA schemes.

7.3.2 Qualitative Evaluation

The GA worked with no domain knowledge about the "optimal" shape of the QA scheme. In contrast, the human subjects possessed superior domain knowledge. For instance, they tried only schemes with high QA effort in the early phases, declining to lower levels at the later phases. It is interesting to note that the solution suggested by the GA also has similar characteristics (see Figure 4). The QA effort starts with a high effort in the early phases, tapering off to a uniform level during the middle of the project. Also, the QA effort increases towards the end of the project. This scheme is consistent with research results [Pressman 2001] that suggest that

- the cost of not detecting errors injected in the early phases can be very high and therefore a high QA effort during these phases is likely to have high pay-off.
- the cost per bug fix during the final stages of the project is very low and therefore increasing QA effort during these phases is likely to have high pay-off.

The above comparisons suggest that the GA produces a scheme that not only improves the objective function, but also has desirable characteristics from an implementation perspective. In summary, the GA enhances the usefulness of the dynamics simulator as a DSS by automating the search for an optimal solution. Further, it provides the project manager the ability to control the direction of the search (by specifying appropriate fitness functions with penalty or rewards) based on domain specific knowledge.

The results seem to suggest that the project managers in the DE-A project used more than QA personnel than necessary throughout the project, possibly explaining the increased total development costs. It should be noted that a variety of factors beyond cost considerations may explain their behavior. First, the project was a critical for the launch of a satellite system by NASA and serious schedule slippages were not permitted. In fact, all software was required to be accepted and frozen three months before the launch date. As the deadline approached, the project managers were under pressure to reduce the chances of delays by adding more personnel to the QA task, even at the risk of increasing the total project costs. Also, the management had underestimated the project's estimated schedule (time for completion) but was not inclined to change this estimate until very late in the project's lifecycle. Instead, additional workforce was added to various activities (possibly at more than necessary levels) to meet this schedule. Such behavior is typical for political reasons [Demarco and Boehm 1998] suggesting that the optimal schemes may at times not get implemented for considerations other than project cost.

8. CONCLUSIONS

The GA when used with the system dynamic simulation has proved to be effective decision support mechanism for arriving at an optimal QA scheme. The dynamic simulator is similar to a flight simulator for software project management and provides the ability for project managers to experiment with decision variables. Such a DSS may help the project manager understand repercussions of various assumptions and scenarios on critical outcomes of a project. Further, examination of various solutions may assist in problem redefinition and the identification of constraints on the solution space. Then, a GA can be assigned the task of finding the best solution within those constraints. As Holland advocates in [Holland 1992], the focus of our approach is more on improvement and less on optimization. The dynamic simulation model combined with a GA can be a very powerful tool for solving a variety of similar, complex problems in software project management.

NOTES

[1] In our experiments, the values were set between 10% and 74% as this is considered an appropriate range for the problem.

REFERENCES

Abdel-Hamid, T. (1988), "The Economics of Software Quality Assurance: A Simulation-based Case Study," *MIS Quarterly*, 395-411.

Abdel-Hamid, T. (1993), "Adapting, Correcting, and Perfecting Software Estimates: A Maintenance Metaphor," *IEEE Computer* March, 20-29.

Abdel-Hamid, T. and S. E. Madnick (1991), *Software Project Management*, Prentice-Hall Inc, Englewood Cliffs, NJ.

Basili, V. R. and D. M. Weiss (1985), "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory," *IEEE Transactions on Software Engineering* 11 2, 157-168.

Baskerville, R., L. Levine, J. Pries-Heje, B. Ramesh and S. Slaughter (2001), "How Internet Software Companies Negotiate Quality," *IEEE Computer* 34 5, 51-57.

Beasley, D., D. R. Bull and R. R. Marin (1993), "An Overview of Genetic Algorithms: Part 2, Research Topics," 15 4, 170-181.

De-Jong, K. A. (1999), “Evolutionary computation for discovery,” *Communications of the ACM* 42 11, 51-53.

Demarco, T. and B. W. Boehm (1998), *Controlling Software Projects: Management, Measurement, and Estimates*, Prentice Hall, Englewood Cliffs, NJ.

Dhar, V., D. Chou and F. Provost (2000), “Discovering Interesting Patterns for Investment Decision Making with GLOWER -- A Genetic Learner Overlaid With Entropy Reduction,” *Data Mining and Knowledge Discovery* 4 4, 251-280.

Dunn, R. (1994), "Quality Assurance," In *Encyclopedia of Software Engineering*. J. Marciniak, Ed., John-Wiley, New York, NY, pp. 941-958.

Dworman, G., S. O. Kimbrough and J. D. Laing (1995), “On Automated Discovery of Models using Genetic Programming: Bargaining in a three-agent coalition game,” *Journal of Management Information Systems* 12 3, 97-125.

Forrest, S. (1993), “Genetic Algorithms: Principles of natural selection applied to computation,” *Science* 261, 872-878.

Forrest, S. (1996), “Genetic Algorithms,” *Computing Surveys* 28 1, 77-80.

Forrester, J. W. (1961), *Industrial Dynamics*, The MIT Press, Cambridge, MA.

Gilb, T. (1988), *Principles of Software Engineering*, Addison-Wesley, Reading, MA.

Glass, R. L. (1982), *Modern Programming Practices: A report from Industry*, Prentice-Hall, Englewood Cliffs, NJ.

Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, MA.

Goldberg, D. E. (1994), “Genetic and Evolutionary Algorithms Come of Age,” *Communications of the ACM* 37 3, 113-119.

Goldberg, D. E., K. Deb and J. H. Clark (1992), “Genetic Algorithms, noise, and the sizing of populations,” *Complex Systems* 6 8, 333-362.

Grefenstette, J. J. (1986), “Optimization of control parameters for genetic algorithms,” *IEEE Transactions on Systems, Man, and Cybernetics* 16 1, 122--128.

Holland, J. H. (1992), *Adaptation in Natural and Artificial Systems*, The MIT Press, Cambridge, MA.

Holsheimer, M. and A. Siebes (1994), "Data Mining: The Search for Knowledge in Databases," CS-R9406, CWI, Amsterdam, The Netherlands.

Koza, J. (1994), *Genetic Programming II: Automatic discovery of reusable programs*, MIT Press, Cambridge, MA.

Koza, J. (2000), “Human-Competitive machine intelligence by means of genetic programming,” *IEEE Intelligent Systems* 15 3, 76-878.

Koza, J. R. (1992), *Genetic Programming: On The Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA.

Koza, J. R., F. H. Bennett, M. Keane and D. Andre (1999), *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann, San Mateo, CA.

Koza, J. R., D. E. Goldberg and D. B. Fogel, Eds. (1996)._*Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA, MIT Press.

Levy, S. (1992), *Artificial Life*, Vintage Books, New York.

Michalewicz, Z. (1996), *Genetic Algorithms + Data Structures = EvolutionPrograms*, Springer-Verlag, New York, NY.

Myers, G. J. (1976), *Software Reliability: Priciples and Practices*, John Wiley & Sons, Inc, New York, NY.

Pressman, R. (2001), *Software Engineering: A Practitioner's Approach*, Fifth Edition, McGraw Hill, New York, NY.

Schraudolph, N. N. and R. K. Belew (1992), “Dynamic Parameter Encoding for Genetic Algorithms,” *Machine Learning* 9, 9-21.

Schraudolph, N. N. and J. J. Grefenstette (1992), "A User's Guide to GAucsd 1.4," CS92-249, CSE Department, University of California, San Diego, CA.

Sholtes, R. M. (1994), "Optimizing System Behavior using Genetic Algorithms," *In Proceedings of the International System Dynamics Conference*, Boston, MA.,pp. 237-247

Srinivas, M. and L. M. Patnaik (1994), “Genetic Algorithms: A survey,” *IEEE Computer* 27 6, 17-26.

Improved Fault-Prone Detection Analysis of Software Modules Using an Evolutionary Neural Network Approach

Robert Hochman[1], Taghi M. Khoshgoftaar[1], Edward B. Allen[2] and John P. Hudepohl[3]

[1]Empirical Software Engineering Lab
Department of Computer Science and Engineering
Florida Atlantic University, Boca Raton, Florida, USA
taghi@cse.fau.edu

[2]Department of Computer Science
Mississippi State University
allen@cs.msstate.edu

[3]Nortel Networks
Research Triangle Park, NC 27709-3478, USA

ABSTRACT

This paper describes an application of genetic algorithms to the evolution of optimal or near optimal backpropagation neural networks for the classification of fault-prone/not fault-prone software modules to support decision making on resource allocation. The algorithm treats each network in a population of neural networks as a potential solution to the optimal classification problem. Variables governing the learning and other parameters and network architecture are represented as substrings (genes) in a machine-level bit string (chromosome). When the neural net population undergoes simulated evolution using genetic operators – fitness-based selection, crossover, and mutation – the average performance increases in successive generations as better-performing neural nets emerge. We found that, on the same data, the classifications obtained were significantly better using a uniform crossover operator when compared with our previous work using traditional one-point crossover. These results are compared with those from the discriminant analysis statistical approach. The latter approach was found to be inferior on our data set. It is suggested that evolutionary neural networks can be used to successfully attack a broad range of data-intensive software engineering problems, where traditional methods have been used almost exclusively. Evolutionary enhancements of traditional methods are also worth considering.

KEYWORDS

Backpropagation, classification, discriminant analysis, fault-prone module, fitness function, genetic algorithm, neural network, simulated evolution, software metrics, uniform crossover.

1. INTRODUCTION

Classification brings order to a problem domain and supports systematic planning and risk-based decision making. Typically, classifying individuals into groups is based on a set of observed characteristics, such as classifying software modules as high or low risk [16]. Membership in a particular group may imply unobserved information about the individual or may indicate or necessitate a specific treatment of the individual. For example, a high-risk software module may be "treated" to more rigorous testing.

In our study, the individuals to classify are software modules taken from a large software development project. Their observed characteristics are software product metrics. We want to be able to predict, from a vector of attribute values for a module, which modules merit greater attention by virtue of being fault-prone. When modules are assigned to mutually exclusive groups (fault-prone and not fault-prone), personnel for testing and maintenance can be assigned accordingly. Moreover, some fault-prone modules identified early in one release may be marked for redesign in the next release before they lead to the growth of a code base too brittle to profitably maintain or enhance. The ultimate goal in this informed allocation of software project resources is to contain costs and maintain schedules with minimal impact on software quality.

Artificial neural networks [17, 18] and multivariate statistical methods, such as discriminant analysis [16, 20], have been commonly used to build computer models which perform this classification process. In discriminant analysis and other statistical methods, few adjustments in the form of configurable parameters are available when attempting to optimize the model. Neural networks, however, offer so many adjustments that optimizing the model by the trial and error approach is intractable, and in this domain of software engineering, rules of thumb are not yet validated. Fortunately, neural networks lend themselves to hybridization with evolutionary computation methods (general purpose search and optimization algorithms inspired by biological evolution), which automate global traversals through the search space of candidate neural networks. Some researchers have developed genetic algorithms for designing and optimizing neural network architectures [8, 25].

In previous work [11, 12], we found that significantly better performing neural networks for detection of fault-prone modules can be obtained using the classical genetic algorithm than by manual trial and error (as is common practice), with great savings in time and effort and greater confidence in the optimality of the results. In the present study, we investigate techniques for improving and extending genetic algorithms to achieve increased accuracy, and we compare our results with those from discriminant analysis on the same data, and the difference in results is tested for statistical significance.

In this paper, the term *evolutionary neural network* (ENN) will be used to signify an artificial neural network whose architectural, learning, and training parameters have been optimized by evolutionary computation, usually a genetic algorithm (GA). Hence an ENN is the end product of simulated evolution on a population of static competing solutions selected for superior performance. A similar term is used in a less restricted sense elsewhere [32].

The following sections cover methods, results and conclusions. In the next section, we describe the techniques used for preparing data, neural networks, genetic algorithms, discriminant analysis, and comparing two proportions for significance. We then present the case study, which classifies software modules as fault-prone or not. Finally, conclusions and directions for future research are presented.

2. EXPERIMENTAL METHODS

In this study, we have measurements on a sample of n software modules, using a set of m software metrics, $X_j, j = 1, \cdots, m$. Let $\mathbf{X}$ be the $n \times m$ matrix of measurements. The following is a summary of our methodology [14].

1. Transform the raw data.
    a) Standardize measurements to a mean of zero and a variance of one for each metric.
    b) Perform principal components analysis on the standardized product metrics to produce *domain metrics*.
2. Prepare data sets.

    Because we had data from only one project which did have a sufficiently large number of modules for meaningful statistical results, we impartially split the data into *training*, *test*, and *validation* data sets. The *fit* data set consists of all modules not in the *validation* data set.
3. Develop models.
    a) Develop a best evolutionary neural network model based on the *training* and *test* data sets.
    b) Develop a nonparametric discriminant analysis model based on the *fit* data set.
4. Predict the class of each module in the *validation* data set using the discriminant model and the best neural network model.

5. Evaluate the accuracy of the models by comparing proportions of predictions to proportions of actual values, using a test for statistical significance.

A classification model is evaluated by the portions of modules that are assigned to the wrong class. *Type I* misclassifications classify *not fault-prone* modules into the *fault-prone* group. *Type II* misclassifications classify *fault-prone* modules into the *not fault-prone* group. The overall misclassification rate is all misclassifications divided by the number of modules.

2.1. Data Preparation

To render the data manageable and usable by the neural net program, and to make the training time tractable, several data preparation steps are followed.

2.1.1. Standardization

Because software metrics have a variety of units of measure, any modeling methodology must reconcile the units of measure. We standardize software metric data, so that the unit of measure becomes one standard deviation.

Let the population mean of the j^{th} metric be estimated by the average, $\bar{x}_j$, of a set of measurements, $x_{1j}, \ldots, x_{nj}$, and let the population standard deviation of the j^{th} metric be estimated by the sample standard deviation, s_j. A *standardized* metric is defined as

$$Z_j = \frac{X_j - \bar{x}_j}{s_j} \tag{1}$$

for all $j = 1, \cdots, m$. Thus, all Z_j have a mean of zero and a variance of one. Let $\mathbf{Z}$ be the $n \times m$ matrix of standardized measurements where z_{ij} is an element, each row corresponds to a module, and each column is a standardized metric.

2.1.2. Principal components analysis

When there is some degree of correlation among the input variables, as is usually the case with software metrics, principal components analysis (PCA) enables one to reduce the dimensionality of the input data without significant loss of information by clarifying the directions of greatest variation in the data set. The resultant gain in speed and ease of training makes the use of this classic multivariate statistical technique worthwhile.

Software product metrics are often highly correlated with one another, because they measure related attributes of the software. Principal components analysis is a technique for transforming multivariate data into variables that are not correlated, and thus, they result in a more robust model [31]. When the original data are software metrics, we call the new principal component variables *domain metrics*.

Principal components analysis is also a data reduction technique. Given m product metrics, a stopping rule chooses $p << m$ domain metrics, and ignores the remaining domain metrics because they have insignificant variation across the data set [31].

Recall that we have m product measurements on each of n modules. Principal components analysis performs the following calculations, given an $n \times m$ matrix of standardized metric data, $\mathbf{Z}$ [31].

1. Calculate the covariance matrix, $\mathbf{\Sigma}$, of $\mathbf{Z}$.
2. Calculate eigenvalues, λ_j, and eigenvectors, $\mathbf{e}_j$, of $\mathbf{\Sigma}$, $j = 1, \ldots, m$.
3. Reduce the dimensionality of the data. In this study, we chose to explain at least 95% of the total variance of the original standardized metrics. Choose the minimum p such that. $\sum_{j=1}^{p} \lambda_j / m \geq 0.95$.
4. Calculate a standardized transformation matrix, $\mathbf{T}$, where each column is defined as

$$t_j = \frac{\mathbf{e}_j}{\sqrt{\lambda_j}} \quad for \ j = 1, \cdots, p \tag{2}$$

5. Calculate domain metrics for each module, where

$$D_j = \mathbf{Z}\mathbf{t}_j \tag{3}$$

$$\mathbf{D} = \mathbf{Z}\mathbf{T} \tag{4}$$

The end result is an $n \times p$ matrix of domain metric data, $\mathbf{D}$, where each domain metric, D_j, has a mean of zero and a variance of one. Since they are orthogonal, the domain metrics are suitable as independent variables.

We used the principal components analysis feature in the SAS statistical package. Further mathematical details are available in statistics texts [22].

2.1.3. Scaling

The transformed data, $\mathbf{D}$, are further processed with a scaling function for compatibility with the neural network. Each value, d_{ij}, is thereby mapped into the closed interval $[0,1]$. For each dimension of the vector, the minimum, $\min_j$, and the maximum values, $\max_j$, of the data set are determined. For neural network input, in_{ij},

$$in_{ij} = \frac{d_{ij} - \min_j}{\max_j - \min_j} \qquad (5)$$

The not fault-prone class is encoded as class 1 and the fault-prone class is encoded as class 2.

2.1.4. Data splitting

We impartially split the data into two sets with similar fault distributions, one twice the size of the other (or nearly so). For the discriminant analysis experiments, the larger set is taken to be the *fit* set and the smaller serves as the *validation* set. Similarly, for the neural network experiments, the smaller of the two becomes the *validation* set, identical with the validation set in discriminant analysis work. However, the larger of the two sets is further split randomly into a *training* set and a *test* set.

In our study, the network is trained on one file and run on two other files of similar data. The files required for the method used here are:

- The *training* set, from which the independent variables and the actual class are read and fed through the network to create the weight set for the model.
- The *test* set, which is used to monitor the ability to interpolate data while the model is being developed. It allows one to avoid overfitting.
- The *validation* set, which is introduced only when the model is finished to verify its ability to generalize to new data from the same environment.

The test set and the validation set should approximate the distribution of classes expected to be found in the task environment. The training set, however, is evenly balanced between fault-prone and not fault-prone patterns to facilitate training by preventing fixation on the predominant class. All of the observations in the less numerous class are concatenated with an equal number pseudorandomly selected from the more numerous class.

2.2. Neural Network

The neural-net classification model for this study is a supervised learning, multilayer, feedforward backpropagation network. This is a rather standard form of neural network. It is well suited to modeling complex non-linear functions, including classification functions. A more thorough treatment is available in many excellent references on various levels [5, 9, 10].

As illustrated in Figure 1, the architecture of this system consists of connected nodes, or processing units, organized hierarchically into layers, and, except in the last layer, the units of one layer are fully connected to all of the nodes in the next forward layer. The connections communicate weighted real-number values from one unit to another. In the feedforward neural network paradigm, no connections occur within layers, across layers, or in the backward direction: connections are made between units in adjacent layers in only one direction.

In our neural net classifier, the activation states of the output units represent the degree of membership in each class. The output with the larger value indicates the predicted class.

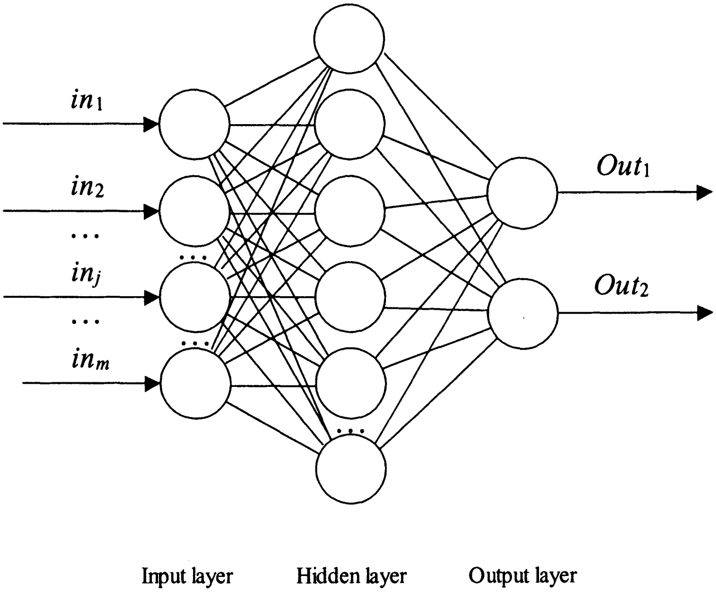


Figure 1. Multilayer feedforward neural network.

Each neuron-to-neuron connection has a variable weight quantifying the connection strength. Let W_{ij} be the connection weight between neurons i and j. Each input-layer neuron receives its input directly from a single input variable. Let in_j be the value of the input variable for input-layer neuron j. Then the summed input at a neuron j is given by

$$Net_j = \begin{cases} in_j & \text{if } j \text{ is an input - layer neuron} \\ \sum_{i=1}^{n} W_{ij} Out_i - \theta_j & \text{otherwise} \end{cases} \tag{6}$$

where θ_j is the threshold of neuron j having n neural inputs. The output of neuron j is given by the logistic function

$$Out_j = \begin{cases} Net_j & \text{if } j \text{ an input - layer neuron} \\ 1/(1+e^{\beta\, Net_j}) & \text{otherwise} \end{cases} \tag{7}$$

where β adjusts the gain of the function.

The network learns by finding a vector of interconnection weights that minimizes its error on the training data set, a data set having known inputs and known outputs. After the connection weights have been selected, the network can predict the outputs for data having known inputs and unknown outputs. These actions, learning and predicting, occur in the two phases of neural network activity.

During training on a data set having N observations, a network with M output layer neurons attempts to find a vector of connection weights, $\mathbf{W}$, that minimizes

$$E(\mathbf{W}) = \sum_{p=1}^{N} \sum_{i=1}^{M} (d_{pi} - Out_{pi})^2$$

where d_{pi} and Out_{pi} are, respectively, the desired and actual output values of i^{th} output layer neuron on the p^{th} observation. In our study, $M = 2$. In this study, the neural networks achieve this using a backpropagation learning algorithm. At the beginning of the training phase, $\mathbf{W}$ is a random vector. The network iterates through the training data adjusting $\mathbf{W}$. Let $W_{ij}(n)$ be the interconnection strength between neuron i in layer $(l-1)$ and neuron j in layer l after the n^{th} iteration through the training data set. The following relationship adjusts the weights:

$$W_{ij}(n+1) = W_{ij}(n) + \eta Out_i \delta_j + \alpha(W_{ij}(n) - W_{ij}(n-1))$$

where η and α set the learning and momentum rates, respectively, and δ_j gives the error contribution for neuron j. For the output layer,

$$\delta_j = (d_j - Out_j)Out_j(1 - Out_j)$$

The desired output for the hidden layer neurons is not known. However, since the error contribution of the hidden layer neurons propagates to the output layer, backward propagation of error allows estimation of the hidden layer neuron error contributions,

$$\delta_j = (\sum_k \delta_k W_{jk})Out_j(1 - Out_j)$$

where k is the running index for the neurons in layer $(l+1)$.

The algorithm iterates through all of the inputs until the maximum number of iterations is reached. To facilitate training, the order of presentation to the net of the *training* set observations is randomized with respect to class membership.

In our study, no attention is given to whether the net converges to a minimum error level. The theoretical justification for this is given by Lin and Vitter [21] and Blum and Rivest [3], who proved that the training problem for even the simplest neural networks is NP-complete. In practical terms, a test for convergence is difficult to precisely define, would greatly increase running time, and, for the success of the genetic algorithm in our problem domain, is unnecessary. Furthermore, reducing the training error in a neural net to zero is not desirable since this often leads to overfitting: it will perform perfectly on the memorized training set but will not perform well on other similar sets of new data. The optimizing program needs only to determine for each net the best classification (on the test set) within the window from 1 to the specified maximum number of epochs. When the model is incorporated into an application, new data must have the same dimensionality and scaling (or lack of scaling), and the activation function and gain parameters for the neural net must match those of the model (the training parameters used to build the model – learning parameter, momentum, update method for weight adjustment – do not matter in the application stage).

2.3. Genetic Algorithm

The genetic algorithm (GA), inspired by natural evolution, is a global search method which can simplify and automate searches in complex, multimodal spaces. It was developed and formalized by Holland [13]. It was further developed and shown to have wide applicability by Goldberg [7]. Schaffer, *et al.* [30] showed that it could be used to improve the learning ability of neural networks for simple pattern discrimination on a small data set. Evolving the weight set for a neural net classifier with the inverted error as the fitness function has also been studied [27]. Ways of combining GAs with neural networks to form improved hybrid algorithms constitute a major research direction. For a good introduction to GAs and an examination of recent work, see Mitchell [26]. Michalewicz [24] is also an invaluable primer.

GAs can be modified in many nontraditional ways, but they generally have the following computationally simple steps:

1. Initialize a population of individuals $P(0)$ with random gene values; set $i=0$.
2. Evaluate the fitness of each member of $P(0)$ according to specified criteria.
3. If the terminating condition is satisfied, stop.
4. Select according to fitness members of the current generation $P(i)$ as parents.
5. Recombine the genes of selected parents (crossover) to get the members of the next generation $P(i+1)$.
6. Mutate some genes in the members of $P(i+1)$ according to a given mutation probability.
7. Evaluate the fitness of each member of $P(i+1)$; increment i.
8. Go to 3.

Most importantly, a GA operates on populations. A population provides a multitude of potential solutions to a problem, and maintains, in effect, a reservoir of potentially valuable gene combinations. Each individual is a combination of *genes* which characterize it or influence its behavior. The genes occur in a usually fixed-length sequence called a *chromosome*. In each generation, each chromosome is assessed by the fitness function for its value in solving the problem. A chromosome, in this study, is the binary encoding of the blueprint for a neural net, which demonstrates some performance value when it is run on the test data. This performance value is determined by the fitness function.

In the next generation, new solutions are created with crossover and mutation, which operate on chromosomes. Bitwise mutation was implemented, where mutation operates on each bit of an offspring chromosome. After crossover has occurred with a likelihood given by the crossover rate, each bit is flipped or not with a probability equal to the mutation rate. Some mutation is needed to maintain the diversity of the *gene pool* – the collective genetic information in a population. But this should occur at a low rate. With a high mutation rate, the GA would degenerate to no better than a random search; any good building blocks developed would quickly disappear from a highly unstable gene pool.

Crossover is a mean of recombining good genetic building blocks from the most fit individuals into individuals of the next generation. It provides the major driving force for progress in the run of a GA, and the crossover rate is usually set to a high value.

For our prior experiments [11], the traditional *one-point crossover* was used. A crossover point in the chromosome is selected randomly. All genes after that point are swapped in the pair of offspring. For example, let the parents be 111000 and 000111 (to make it easy to see). If the crossover point is 3, then the offspring are 111111 and 000000. This crossover operator suffers from positional bias. The genes at the end of chromosome string will always be exchanged. Using two crossover points (*two-point crossover*) or more (*n-point crossover*), will overcome this drawback to some extent, but sequences of adjacent genes are likely to be exchanged in a crossover. While *uniform crossover* eliminates positional bias, it is the most disruptive to the integrity of the chromosome.

Uniform crossover is a generalization of one-point crossover. In one-point crossover, all bits before and including the bit at the crossover point have probability 0 (no probability) of exchange and all bits after the crossover point have probability 1 (the probability of certainty). In uniform crossover, the number of crossover points is the number of bit positions in the chromosome and each is exchanged with some probability p between 0 and 1. Usually, p is 0.5 (0.5-uniform crossover), but higher values may be better. Of course, when p is 1, no recombination occurs because crossover occurs at every point and the offspring are identical to the parents, the first of the pair to the second parent and the second to the first parent.

Uniform crossover is in some ways analogous to the generalized sets exploited in fuzzy logic, which extends the 0/1 set membership of classical set theory by admitting intermediate values. This analogy suggests another apparently untried variation of one and two-point crossover, which may be worth investigating, in which crossover points are selected as usual but bits after the crossing point are exchanged with a probability intermediate between 0 and 1.

For the software engineering data, selection of individuals for mating is done by the fitness proportionate method. If the fitness of a particular individual a is f_a, then the probability of selection for a in a population of size N is

$$P_a = \frac{f_a}{\sum_{i=1}^{N} f_i}$$

where $\sum_{i=1}^{N} f_i$ is the population fitness, the sum over all individual fitnesses. Thus, an individual is selected in proportion to its contribution to the population fitness.

Tournament selection of size 2 was used in one experiment. In tournament selection of size n, n individuals are randomly selected from the population and the individual with highest fitness among these is used in mating. This selection process is repeated for the number of mates required.

In the fixed population size, generational paradigm used here, while the population size has not been reached, two individuals are selected as parents to produce a pair of offspring to replace the parents in the new generation. Selecting the best individual or the best n individuals to copy without modification into the next generation, is known as *elitist selection*. It is helpful in preventing the degeneration of the gene pool. In our experiments, elitist selection was used, with n set to 4. When a run of our GA optimizing program terminates, the four best results are output, thus giving a range of good choices rather than a single, inflexible answer. Typically, one model may give somewhat better class 2 performance at the slight expense of class 1, or vice versa. We have two often conflicting objectives: optimizing the identification rate for each of two classes. So, providing a range of choices to the user is a rational strategy.

The genetic algorithm, itself, is application independent, but in order for it to work in a specific application, two problems must be addressed in terms of the application – the representation problem and the fitness problem. Their solutions are not always straightforward. The following two subsections will explain the solutions developed for our problem domain.

2.3.1. Representation problem

At the machine level, an individual in the population is a string of bits (non-binary alphabets may also be used). To the GA, this string is a chromosome, and algorithmically, an individual is identical with its chromosome. The chromosome is a sequence of genes (bit substrings), whose encoded values (*alleles*) characterize the individual. How to encode the essential attributes of an individual in its chromosome is known as the *representation problem*. Its solution is application-specific.

Since we want to build high-performance classification nets, the genes should encode network characteristics that are controlled by parameters which influence network performance. For the backpropagation algorithm, probably the most important parameters are the number of hidden layers, the number of units in each layer, the learning rate η, and the momentum α. To these, we added the update method (continuous or periodic updating of the weights during training) and the gain, the factor in the activation function. For most of these experiments, the number of hidden layers was fixed at one, since our empirical evidence [11,12] showed no significant advantage in more than one layer in this domain.

Half of the results below used a chromosome for an individual that was 33 bits in length and had the following gene sequence:

- Number of hidden layers – 2 bits.
  (Not used because the number of layers was fixed at 1)
- Number of units in hidden layer (maximum 128) – 7 bits.
- The learning rate (range 0 to 1) – 8 bits.
- The momentum (range 0 to 1) – 8 bits.
- Update (continuous or periodic) – 1 bit.
- Gain (range 0 to 1) – 7 bits.

The other half of the results below used a 41 bit-length chromosome. An extra 8-bit gene was added to encode separate learning rates on the the input layer to hidden layer error and on the hidden layer to output layer error. As a consequence, the size of the binary search space was increased by a factor of 256.

Each gene encodes a real-valued or an integer-valued parameter. The integers are decoded by the GA from their binary number representations into decimal integers for use by the program. In the case of the number of hidden layers or the number of units in a hidden layer, the value is augmented by 1, since a zero value for these variables is not permitted in the neural net algorithm. Update needs only one bit to represent it in the chromosome since it is coded as a Boolean variable – continuous or not continuous (periodic). The genes for the learning rate (η), the momentum (α), and the gain encode real numbers. Representation of real numbers in a string of binary values entails some difficulties. In our representation, eight-bit substrings for η and α were coded as binary fractions and decoded to decimal fractions with division by 255. The gain was represented as a seven-bit substring and divided by 127 to obtain a decimal fraction.

A real-number encoding was also developed for comparison tests. In this representation, the chromosome is an array of real numbers. Each of the above mentioned parameters is represented by a gene whose value is a real number between 0 and 1. If, however, the range of permissible values for the parameter is between 0 and α, the real-number value of the gene is multiplied by α. If the parameter is Boolean, the value of its gene is tested to determine whether or not it is less than 0.5. The comparable real-number chromosome has a length of 6. Crossover for real-number chromosomes can be implemented in the same way as for binary chromosomes – by swapping corresponding genes (real numbers rather than binary digits) from the parents in opposite ways for the two offspring – or the genetic information from the selected parents can be recombined by taking the mean of the values of corresponding genes. In this latter case, two parents have just one offspring. Mutation for a gene with real value α, $0 \leq \alpha \leq 1$, is implemented by either incrementing the gene value by a random percentage of $1-\alpha$ or, with equal probability, decrementing by a random percentage of α.

2.3.2. Fitness problem

A fitness function is needed to rank the members of a generation for selection. What the fitness function actually measures (often not what the researcher expects it to measure) is maximized. Since what it measures is dependent on an application-specific decoding of chromosomes, the fitness function must be specifically designed for the application.

Most work in evolving neural networks has made the complement or inverse of the error used for weight readjustment the measure of fitness [1, 4, 19, 25, 27]. For our problem, however, the optimization task is complicated by the fact that minimization of the learning error is not sufficient. The magnitude of the error does not clearly indicate how successful the net is in separating classes with different frequencies since an undesirable reduction in the the accuracy for one class can nevertheless greatly reduce the total error.

A neural network's degree of success with respect to class separation is readily determined from the *confusion matrix* it produces for a data set. For the two-category problem considered here, this is a 2×2 matrix of integers. The first row contains the number of correctly classified modules of class 1 (not fault-prone) and the number of incorrectly classified modules of class 1. The second row contains the number of incorrectly classified modules of class 2 (fault-prone) and the number of correctly classified modules of class 2 in that order. The position of a number in the matrix encodes its semantic interpretation.

A perfect classification would have zeroes on the minor diagonal and the total numbers of class 1 and class 2 modules on the major diagonal, as follows:

$$\begin{bmatrix} 2004 & 0 \\ 0 & 320 \end{bmatrix}$$

Here, 2004 means the total number of class 1 and 320 the total number of class 2 patterns (the actual numbers for the validation set used in the neural network experiments and for discriminant analysis). In the N-category problem, $\mathbf{A}$ is an $N \times N$ matrix whose entry A_{ij} (in the i^{th} row, j^{th} column of $\mathbf{A}$) is the number of times a pattern in class i has been identified as a member of class j. For our problem, if N_{11} and N_{22} denote the number of correct class 1 and class 2 identifications, respectively, and N_{12} and N_{21} the number of class 1 patterns identified as class 2 and the number of class 2 patterns identified as class 1, respectively, then

$$\mathbf{A} = \begin{bmatrix} N_{11} & N_{12} \\ N_{21} & N_{22} \end{bmatrix}$$

is the general confusion matrix which is output by a neural net routine for a set of patterns. This matrix is the input to the fitness function f.

In prior work [11], several different fitness functions were devised, tested, and evaluated. The fittest of these is the surviving fitness function used in the present experiments. It is as follows:

$$f(\mathbf{A}) = \begin{cases} 0.1 & : \quad p_1 \le P_1 \quad or \quad p_2 \le P_2 \\ 0.1 + (p_1 - P_1) + (p_2 - P_2) & : \quad p_1 > P_1 \quad \& \quad p_2 > P_2 \end{cases}$$

where p_1 and p_2 are the model's identification rates for class 1 and class 2 respectively, and P_1 and P_2 are the corresponding minimum cutoff rates. When the identification rate for either class is less than or equal to the minimum cutoff rates for the class the individual is penalized with a fitness value of 0.1. Otherwise, the fitness value is 0.1 plus the sum of the amounts by which each class identification rate exceeds its specified minimum rate – *i. e.*, $\sum_{i=1}^{2}(p_i - P_i)$. The minimum fitness is 0.1 in order not to exclude the poorest performers from the gene pool. They have a contribution to make to the diversity of the gene pool and have a nonzero albeit small chance of being selected for reproduction. For our experiments, P_1 ranged from 0.30 to 0.60, and P_2 ranged from 0.72 to 0.75. In many cases, P_1 was set to 0.40 and P_2 to 0.74. Higher values limit the gene pool but generally produce higher performing individuals. Other data sets may require different values.

2.3.3. Implementation

The computational complexity of the code for the genetic optimizer (see Figure 2) is of the order of magnitude of the product of three variables: the maximum number of generations (G), the population size (P), and the aggregate complexity (A) – in big Oh notation, $O(GPA)$. The aggregate complexity A is the complexity of the backpropagation algorithm and its input. It depends on E, the number of epochs (the amount of training), the connectivity of the network (which depends largely on the number of layers as well as the number of units in each layer), and the complexity of the data – the size of the training set, the size of the test set, and the dimensionality of the pattern space. The connectivity must be summed over the variety of architectures in each generation and that sum must itself be summed over the total number of generations. The bit length of the chromosome will also have an effect. Thus, the running time on our Unix-based workstations for experiments of modest size (say, $G = 30$, $P = 10$, $E = 100$) was hours. The code will be most effective when ported to a distributed environment (a cluster of workstations), or a massively parallel machine, with one processor allocated for each neural net in the population.

```
for initial neural net (nn) population
  initialize fixed parameters
  randomly configure other nn params
end for (initial nn population)
set fitness of best nn to zero
for 1 to maximum number of generations
  for 1 to population size
   build nn from its parameters
   set fitness of nn to zero
   for 1 to maxepochs by stepsize
     for stepsize epochs train nn
     run on test data set
     generate the confusion matrix
     compute fitness
     if fitness > previous fitness of nn
      update fitness of nn
      if fitness > fitness of best nn
        update best nn record
       endif ( > fitness of best nn)
     endif (> previous fitness of nn)
   end for (training epochs)
  end for (population)
  using genetic operators, generate
   new population from old population
  replace old pop. with new population
end for (generations)
evaluate the best nn on validation data set
```

Figure 2. Pseudocode for the genetic optimizing program.

However, for a single-processor machine, some time-saving strategies can be exploited. These include running the inner loop for an initial number of iterations without computing the evaluations and comparisons for the best neural net. This can usually be done without any sacrifice because there is a latent, initial formation stage in which the only strategy the net exhibits is to identify all patterns as belonging to one category. The initial step size for this study was often set to 30. Another way to save some time is to skip the inner loop if the offspring neural net has not undergone crossover and mutation and is a copy of one of the parents. It would also be a copy if it is the result of elitist selection. In these cases, the fitness is already known (it is the fitness of the parent), so entering the inner loop is not necessary.

If the training set is sufficiently large and representative, it can often be reduced by uniformly random selection to half its size without loss of essential information. The training set initially used was reduced in this way for the present study – 850 patterns were cut by random selection to 425 patterns. If the stepsize for the inner loop is increased the running time will be shortened. But in these experiments this feature was little used. Probably, the possibility of missing good results with stepsize greater than one becomes negligible only when the number of epochs needed for convergence is very large.

2.4. Discriminant Analysis

Discriminant analysis is a multivariate statistical modeling technique for estimating classification. It can determine with some accuracy to what extent separation into predefined classes is possible for an observation sample with given metrics. As a tool for making decisions about group membership, it is available in many modern statistical packages. We use the nonparametric discriminant analysis tool in the SAS package [29]. We chose the "nonparametric" form of discriminant analysis because density functions were unlikely to be normally distributed [14]. Estimated density functions for two mutually exclusive classes are computed using a normal kernel function on the vectors of independent variables, Bayesian posterior probabilities of membership in a particular class, and the *a priori* probability distributions of the labeled classes in the *fit* data set from which the discriminant model is built [16]. One configurable parameter, a smoothing parameter λ, exists for attempting to minimize classification error over a number of runs with different λ values.

2.4.1. Stepwise discriminant analysis

Recall that our purpose was to classify modules as belonging to either the *not fault-prone* group or the *fault-prone* group. We used *stepwise discriminant analysis* model selection at the 5% significance level to choose the domain metrics, D_j, that should be included as independent variables in the discriminant model [31].

Variables are entered into the model in an iterative manner, based on an F test from analysis of variance which is recomputed for each change in the current model. Begin with no variables in the model. Add the variable not already in the model with the best significance level, as long as its significance is better than the threshold (5%). Then remove the variable already in the model with the worst significance level, as long as its significance is worse than the threshold (5%). Repeat these steps until no variable can be added to the model. The final result is a subset of D_j , $j = 1, \cdots, p$, that are significantly related to the module class.

2.4.2. Estimating parameters

We applied nonparametric discriminant analysis, a standard statistical technique, to predict the membership of each module in the *not fault-prone* group (G_1) or the *fault-prone* group (G_2). We defined the selected domain metrics, D_j, as independent variables, and the group membership was given. We estimated a *discriminant function* based on the *fit* data set [31].

Consider the following notation. Let $\mathbf{d}_i$ be the vector of the i^{th} module's independent variables, and let n_k be the number of modules in group G_k , $k = 1, 2$. Let $\mathbf{S}_k$ be the covariance matrix for all samples in G_k, and let $|\mathbf{S}_k|$ be its determinant. Let $f_k(\mathbf{d}_i)$ be the multivariate probability density giving the probability that a module, $\mathbf{d}_i$, is in G_k, and let $\hat{f}_k(\mathbf{d}_i \mid \lambda)$ be an approximation of $f_k(\mathbf{d}_i)$, where λ is a parameter. From a Baysian probability viewpoint, let π_k be the prior probability of membership in G_k. We choose the prior probability, π_k, to be the proportion of *fit* modules in G_k.

Since the density functions, f_k, are not likely to conform to the normal distribution, we use nonparametric discriminant analysis. Let λ be a smoothing parameter in this context. We select the multivariate normal kernel on vector $\mathbf{u}$ with modes at $\mathbf{v}$. This is the most commonly used kernel, and has been studied the most mathematically.

$$K_k(\mathbf{u} \mid \mathbf{v}, \lambda) = (2\pi\lambda^2)^{-n_k/2} |\mathbf{S}_k|^{-1/2}$$

$$\exp(\quad (-1/2\lambda^2)(\mathbf{u} - \mathbf{v})' \mathbf{S}_k^{-1} (\mathbf{u} - \mathbf{v}) \quad) \tag{8}$$

Let $\mathbf{d}_{kl}$, $l = 1, \cdots, n_k$ be a vector of independent variable values for the l^{th} observation in group G_k. The estimated density function is given by the multivariate kernel density estimation technique.

$$\hat{f}(\mathbf{d}_i \mid \lambda) = \frac{1}{n_k} \sum_{l=1}^{n_k} K_k(\mathbf{d}_i \mid \mathbf{d}_{kl}, \lambda \quad) \tag{9}$$

The estimated discriminant function is given by

$$\text{Assign } \mathbf{d}_i \text{ to} \begin{cases} G_1 & \text{if } \dfrac{\hat{f}_1(\mathbf{d}_i \mid \lambda)}{\hat{f}_2(\mathbf{d}_i \mid \lambda)} > \dfrac{\pi_2}{\pi_1} \\ G_2 & \text{otherwise} \end{cases} \tag{10}$$

This classification rule minimizes the total number of misclassifications [31].

2.5. Comparing Two Proportions For Significance

The following statistical test was used for comparing two proportions [33].

Let $\hat{p}_1 = X_1 / n_1$ and $\hat{p}_2 = X_2 / n_2$ be the estimates of two proportions, p_1 and p_2, where X_i is a count for a sample of size n_i. If we want to test the hypothesis $H_0 : p_1 = p_2$, with alternate hypothesis $H_A : p_1 > p_2$ then

$$Z = \frac{\hat{p}_1 - \hat{p}_2}{\sqrt{\frac{\overline{p}\overline{q}}{n_1} + \frac{\overline{p}\overline{q}}{n_2}}}$$

where

$$\overline{p} = \frac{X_1 + X_2}{n_1 + n_2} = \frac{n_1\hat{p}_1 + n_2\hat{p}_2}{n_1 + n_2},$$

and

$$\overline{q} = 1 - \overline{p} .$$

When $n_1 = n_2$, as in our case, then we get the following reductions:

$$Z = \frac{\hat{p}_1 - \hat{p}_2}{\sqrt{2\frac{\overline{p}\overline{q}}{n}}},$$

where n is the common sample size, and

$$\overline{p} = \frac{\hat{p}_1 + \hat{p}_2}{2} .$$

Assuming roughly normal distributions for $\hat{p}_1$ and $\hat{p}_2$, we can determine the proportion of the normal curve which is greater than or equal to the computed value of Z. This proportion is the level of significance. It is most conveniently found in a table of proportions of the normal curve (one-tailed) by looking up the proportion corresponding to the Z value. If the level of significance is less than a specified limit (usually 5%), the null hypothesis is rejected.

3. CASE STUDY

The software engineering data used in this study was derived from a telecommunications system with about 12 million lines of code written in a Pascal-like proprietary language. About 7000 modules, all of which have undergone some revision since their prior releases, were the source for the data set. In this context, a module is a set of source files designated as such by designers.

The source code was measured by a version of the Datrix metric analyzer [2] customized for the proprietary programming language of the subject software. The metric data was collected for another purpose, and was subsequently made available for this study. The nine metrics listed in Table 1 were selected for study from about fifty metrics collected by the metric analyzer. Selected metrics were limited to those that could be derived from design and pseudocode documentation and to those with properties suitable for modeling [6,15]. These metrics can be collected from pseudocode or source code. They are drawn (at least, theoretically) from call graphs (to measure the connectivity and interdependency of modules) and control-flow graphs (to measure the logical complexity of modules).

The control-flow graph terminology, although standard, may require some explanation here. A *vertex* of a flow graph (for the design of a procedural computer program) is a conditional (if-then) or a sequential declarative (executable) statement, and the transfer of control from one vertex to another is called an *arc* (represented graphically as an arrow). A *loop* is a control structure having a cycle of control from a vertex (usually containing a conditional test) back to itself. A control structure embedded within another control structure (for example, a while loop inside another while loop) is said to have *nesting level* 1. Cyclomatic complexity is a widely known [28] quantitative measure of the logical complexity of a program [23]. For a control flow graph with one entry and one exit, cyclomatic complexity is the number of decision nodes plus one. Zuse presents an in-depth study of its properties [34].

For the training and test sets used to build our model, the nine metrics were matched with the corresponding number of faults reported in the testing of each module (when applying the completed model in normal use, the number of faults will not be available). The frequency distribution of faults was skewed toward the low end – approximately half of the modules had zero reported faults (50.7%) and a few had many faults. The distinction between fault-prone and not fault-prone modules is necessarily subjective, but it ought to be defined to correspond with a practical difference in the software development environment to which it is applied, for example, the percentage of modules to which a manager intends or is able to devote extra attention and resources. In our case, the threshold between these two classes was placed by software development managers at 3. Thus, modules with less than three reported faults were considered not fault-prone (86.3%), and modules with three or more faults were considered fault-prone (13.7%).

Metrics from Call Graph
Unique procedure calls
Total procedure calls
Distinct include files
Metrics from Control Flow Graph
McCabe's cyclomatic complexity
Total loops
Total if-then structures
Total nesting level over all arcs
Total vertices within conditional structures
Total arcs + vertices within loop structures

Table 1. Software metrics used to build classification models.

Given the nine software design metrics, principal components analysis retained four domains under the stopping rule that we retain components which account for 95% or more of the total variance. Table 2 shows the relationship between the original metrics and the domain metrics. Each table entry is the correlation between the metrics. The largest correlation in each row is **bold**.

Metric	D_1	D_2	D_3	D_4
Total vertices within conditional structures	**0.884**	0.313	0.275	0.009
Total nesting level	**0.853**	0.362	0.335	0.012
Total if-then structures	**0.665**	0.601	0.374	0.013
Total procedure calls	0.360	**0.853**	0.307	0.005
Unique procedure calls	0.359	**0.838**	0.367	0.001
McCabe's cyclomatic complexity	0.617	**0.632**	0.416	0.005
Total loops	0.290	0.407	**0.841**	0.046
Total arcs + vertices within loops	0.418	0.316	**0.827**	0.019
Distinct include files	0.003	0.004	0.030	**0.999**
Eigenvalues	2.85	2.69	0.12	1.00
% Variance	31.67%	29.89%	23.56%	11.11%
Cumulative	31.67%	61.56%	85.12%	96.23%

Changed Modules

Table 2. Domain Pattern.

We developed evolutionary neural network models according to our methodology. For each of the classifications of class 2 from 225 correct to 236 correct the best class 1 classifications using ENNs are shown in the confusion matrices in Table 3. The crossover method is identified in column 1. All of the best results, it should be noted, come from models built with uniform crossover. For each entry in Table 3, Table 4 presents in the same order the neural net and GA parameters used to develop the ENN model. The column "max units" is the maximum number of hidden units to which the search space was limited. The column "layers" gives the number of units in the input, hidden, and output layers. The column marked η_2 is the learning rate for connections between the input layer and the hidden layer; the one marked η, the learning rate for connections between the hidden layer and the output layer. Those marked α, "pop", "gen", "Xover", and "mut." are, respectively, the momentum, the population size, the maximum number of generations, the crossover rate, and the mutation rate. In half of these models, the chromosome length was 33, using the representation scheme of prior work [12] – the maximum units was 128 and η_2 was fixed at 0.5. In one of these, α was fixed at 1.0. For all of these models except one, the selection method was fitness proportionate. Tournament selection of size 2 was used to build the model represented by the fifth table entry. The other half of the models used a chromosome length of 41 bits.

The stepwise model selection process found the first three domain metrics significant at the 5% significance level, but not D_4, and therefore, the inputs to the discriminant model were D_1, D_2, and D_3.

The discriminant procedure used the *fit* data set to estimate the multivariate density functions, $\hat{f}_1$, $\hat{f}_2$, and thus, the discriminant function per Equation (10). We empirically determined the kernel density estimation smoothing parameter to be $\lambda = 0.005$. The discriminant function was then used to classify each module in the *validation* data set.

The best classification results obtained using evolutionary neural nets and discriminant analysis are shown in Table 5. To simulate subsystem performance, results were also obtained on randomly created subsets of the validation data.

It is fair to say that, in any large data collection process, some of the data will be *noise* – erroneous data. If we assume that the noise in the data set is sparse and not uniformly distributed, it is likely that proper subsets of the data have less noise. These plausible assumptions may help to explain the better performance of the neural net seen in Table 5. Neural nets, because of the distributed nature of their computation, are tolerant of training noise but perform better when noise is reduced in the task environment. A statistical model, however, can be expected to be accurate only to the extent that the random sample data on which it is built is representative of the population. Therefore, a statistical model built with data containing substantial noise should show no significant improvement in performance tests when the noise is reduced and the probability distribution of the data is altered. This inherent difference in the ability to handle noise may be an explanation for the indisputable difference in performance. The increasingly better neural net results for overall classification on progressively smaller subsets (Table 5) appear to strongly support this hypothesis.

In Table 6 the errors (as real numbers rather than percentages) on class 1, class 2, and overall for each approach on the full set and the random subsets of Table 5 are compared for level of statistical significance. Our null hypothesis is $H_0 : p_1 = p_2$ and the alternate hypothesis is $H_A : p_1 > p_2$, where p_1 and p_2 are the error proportions to be compared for the discriminant analysis and ENN approaches, respectively. The superior performance of ENNs on the full set is most significant on class 2 error and on overall error because, when the discriminant analysis results approach the ENN results on class 1 error, the models diverge considerably in performance on class 2 error.

Real number encoding of the chromosome did not produce significantly better classification results than binary encoding. The results were similar in extensive trials of the two representation schemes – on our data, for our problem, for our real number implementation.

Type of crossover	Count confusion matrix	Percent confusion matrix	$[\%\ \textit{right}\quad \%\ \textit{wrong}]$
0.8-uniform	$\begin{bmatrix}1489 & 515\\ 84 & 236\end{bmatrix}$	$\begin{bmatrix}74.30 & 25.70\\ 26.25 & 73.75\end{bmatrix}$	$\begin{bmatrix}74.23 & 25.77\end{bmatrix}$
0.8-uniform	$\begin{bmatrix}1485 & 509\\ 85 & 235\end{bmatrix}$	$\begin{bmatrix}74.60 & 25.40\\ 26.56 & 73.44\end{bmatrix}$	$\begin{bmatrix}74.44 & 25.56\end{bmatrix}$
0.5-uniform	$\begin{bmatrix}1500 & 504\\ 86 & 234\end{bmatrix}$	$\begin{bmatrix}74.85 & 25.15\\ 26.88 & 73.12\end{bmatrix}$	$\begin{bmatrix}74.61 & 25.69\end{bmatrix}$
0.8-uniform	$\begin{bmatrix}1506 & 498\\ 87 & 233\end{bmatrix}$	$\begin{bmatrix}75.15 & 24.85\\ 27.19 & 72.81\end{bmatrix}$	$\begin{bmatrix}74.83 & 25.17\end{bmatrix}$
0.5-uniform	$\begin{bmatrix}1521 & 483\\ 88 & 232\end{bmatrix}$	$\begin{bmatrix}75.90 & 24.10\\ 27.50 & 72.50\end{bmatrix}$	$\begin{bmatrix}75.43 & 24.57\end{bmatrix}$
0.8-uniform	$\begin{bmatrix}1503 & 501\\ 89 & 231\end{bmatrix}$	$\begin{bmatrix}74.61 & 23.39\\ 27.81 & 72.19\end{bmatrix}$	$\begin{bmatrix}74.61 & 25.39\end{bmatrix}$
0.8-uniform	$\begin{bmatrix}1525 & 479\\ 90 & 230\end{bmatrix}$	$\begin{bmatrix}76.10 & 23.90\\ 28.12 & 71.88\end{bmatrix}$	$\begin{bmatrix}75.52 & 24.48\end{bmatrix}$
0.5-uniform	$\begin{bmatrix}1524 & 488\\ 91 & 229\end{bmatrix}$	$\begin{bmatrix}76.05 & 23.95\\ 28.44 & 71.56\end{bmatrix}$	$\begin{bmatrix}75.43 & 24.57\end{bmatrix}$
0.8-uniform	$\begin{bmatrix}1528 & 480\\ 92 & 228\end{bmatrix}$	$\begin{bmatrix}76.25 & 23.75\\ 28.75 & 71.25\end{bmatrix}$	$\begin{bmatrix}75.56 & 24.44\end{bmatrix}$
0.5-uniform	$\begin{bmatrix}1540 & 464\\ 93 & 227\end{bmatrix}$	$\begin{bmatrix}76.85 & 23.15\\ 29.06 & 70.94\end{bmatrix}$	$\begin{bmatrix}76.03 & 23.97\end{bmatrix}$
0.8-uniform	$\begin{bmatrix}1551 & 453\\ 94 & 226\end{bmatrix}$	$\begin{bmatrix}77.40 & 22.60\\ 29.38 & 70.62\end{bmatrix}$	$\begin{bmatrix}76.46 & 23.54\end{bmatrix}$
0.5-uniform	$\begin{bmatrix}1566 & 438\\ 95 & 225\end{bmatrix}$	$\begin{bmatrix}78.14 & 21.86\\ 26.69 & 70.31\end{bmatrix}$	$\begin{bmatrix}77.07 & 22.93\end{bmatrix}$

Table 3. The Best ENN Classifications.

				Neural net parameters					Genetic parameters			
Max Units	Layers			gain	η	η_2	α	epochs	pop	Gen	Xover	Mut.
12	4	6	2	.0.354331	0.015686	0.384314	0.682353	85	24	24	0.90	0.09
9	4	6	2	0.220472	0.090196	0.137255	0.701961	56	16	20	0.90	0.09
12	4	9	2	0.007874	0.301961	0.850980	0.835294	40	16	3	0.90	0.09
128	4	2	2	0.637795	0.266667	0.500000	1.000000	36	16	20	0.95	0.09
128	4	92	2	0.535443	0.047244	0.500000	0.535433	64	16	20	0.90	0.09
12	4	8	2	0.086614	0.125490	0.839216	0.545098	51	24	24	0.90	0.09
128	4	94	2	0.800000	0.031373	0.500000	0.533333	66	16	20	0.90	0.09
128	4	53	2	0.385837	0.078431	0.500000	0.596078	45	10	10	0.90	0.09
10	4	2	2	0.062992	0.211765	0.500000	0.564706	48	16	20	0.90	0.09
12	4	6	2	0.039370	0.811765	0.184314	0.701961	54	16	3	0.90	0.09
128	4	59	2	0.173238	0.141176	0.500000	0.654902	43	12	10	0.90	0.09
128	4	122	2	0.070876	0.501961	0.500000	0.611765	54	16	20	0.90	0.09

Table 4. NN and GA parameters for best ENN models.

Number of cases	Counting confusion matrix	Percent confusion matrix	$\left[\%\ \mathit{right} \quad \%\ \mathit{wrong}\right]$
The evolutionary neural network results			
2324	$\begin{bmatrix}1521 & 483\\ 88 & 232\end{bmatrix}$	$\begin{bmatrix}75.90 & \mathbf{24.10}\\ \mathbf{27.50} & 72.50\end{bmatrix}$	$\mathbf{[75.43 \quad 24.57]}$
1162	$\begin{bmatrix}776 & 227\\ 39 & 120\end{bmatrix}$	$\begin{bmatrix}77.37 & 22.63\\ 24.53 & 75.47\end{bmatrix}$	$[77.11 \quad 22.89]$
581	$\begin{bmatrix}395 & 112\\ 19 & 55\end{bmatrix}$	$\begin{bmatrix}77.91 & 22.09\\ 25.68 & 74.32\end{bmatrix}$	$[77.45 \quad 22.55]$
290	$\begin{bmatrix}191 & 53\\ 7 & 39\end{bmatrix}$	$\begin{bmatrix}78.28 & 21.72\\ 15.22 & 84.78\end{bmatrix}$	$[79.31 \quad 20.69]$
145	$\begin{bmatrix}102 & 22\\ 4 & 17\end{bmatrix}$	$\begin{bmatrix}82.26 & 17.74\\ 19.05 & 80.95\end{bmatrix}$	$[82.07 \quad 17.93]$
The discriminant analysis results			
2324	$\begin{bmatrix}1444 & 560\\ 126 & 194\end{bmatrix}$	$\begin{bmatrix}72.06 & \mathbf{27.94}\\ \mathbf{39.38} & 60.62\end{bmatrix}$	$\mathbf{[70.48 \quad 29.52]}$
1162	$\begin{bmatrix}750 & 253\\ 58 & 101\end{bmatrix}$	$\begin{bmatrix}74.78 & 25.22\\ 36.48 & 63.52\end{bmatrix}$	$[73.24 \quad 26.76]$
581	$\begin{bmatrix}362 & 145\\ 29 & 45\end{bmatrix}$	$\begin{bmatrix}71.40 & 28.60\\ 39.19 & 60.81\end{bmatrix}$	$[70.05 \quad 29.95]$
290	$\begin{bmatrix}188 & 56\\ 16 & 30\end{bmatrix}$	$\begin{bmatrix}77.05 & 22.95\\ 34.78 & 65.22\end{bmatrix}$	$[75.17 \quad 24.83]$
145	$\begin{bmatrix}92 & 32\\ 11 & 10\end{bmatrix}$	$\begin{bmatrix}74.19 & 25.81\\ 52.38 & 47.62\end{bmatrix}$	$[70.34 \quad 29.66]$

Table 5. The experimental results.

set no.	class	DA error	ENN error	no. of cases	significance
1	1	0.2794	0.2410	2004	0.0028
2	1	0.2522	0.2423	1003	0.3050
3	1	0.2860	0.2158	507	0.0049
4	1	0.2295	0.2213	244	0.4129
5	1	0.2581	0.2283	124	0.2912
1	2	0.3938	0.2750	320	0.0007
2	2	0.3648	0.2830	159	0.0594
3	2	0.3919	0.2500	74	0.0322
4	2	0.3478	0.2391	46	0.1271
5	2	0.5238	0.1667	21	0.0075
1	overall	0.2952	0.2457	2324	0.0001
2	overall	0.2676	0.2478	1162	0.1379
3	overall	0.2995	0.2203	581	0.0010
4	overall	0.2483	0.2241	290	0.2451
5	overall	0.2965	0.2207	145	0.0708

Table 6. Significance tests for discriminant analysis and ENNs.

4. CONCLUSIONS AND FUTURE DIRECTIONS

Clearly, the discriminant analysis model did not outperform the ENN model on the data sets of the present study. The discriminant model is limited by one degree of freedom (the smoothing parameter) for adjustment in the range of results. The ENN model, however, has numerous degrees of freedom including, but not limited to, the population size, the number of generations, the crossover rate, the mutation rate, and the minimum acceptance rates in the fitness function. Ultimately, the comparison is between a deterministic statistical algorithm and a directed stochastic one. The latter has the advantage of adaptability.

The results shown in Table 3, for which uniform crossover was used, are all improvements over those published previously [11]. For example, the best previous class 1 identification for 232 correct identifications was 1499. The best corresponding result in this study is 1521. These results establish the suitability of ENNs as a software engineering tool.

While our earlier work dealt with the curse of parametrization in designing neural nets, the present work has explored the blessings of higher level parametrization for optimizing the design of neural nets.

Neural net design requires decisions about the number of layers, the number of hidden units, the learning parameter, the momentum, the update method, the transfer function, to name a few. All of these make the search space for the optimum neural network very large, making the search by manual control intractable. When this is left to a genetic algorithm, the search is automated. The solutions it produces may well be satisfactory for many managerial purposes.

In the training and test data, the accuracy of the number of faults depends on consistent human reporting practices. The fault reports are imperfect in that it is unclear to what extent standardized fault definitions were consistently applied. For example, two reported faults may be caused by the same piece of code. The accuracy of the metrics data depends on the correctness of the metrics analyzer software and the consistency between versions. Since only synthetic software engineering problems come with perfect data, there is a need for real-world methods of learning from imperfect data (noisy or incomplete or inconsistent data). The results obtained show what is possible using ENNs – even with imperfect data.

In the software engineering data set, the data are partly human-supplied, partly automatically generated discrete variables characterizing objects which are human-contrived logical constructs. These discrete quantities are linked to fault counts which are used to partition the data set to accord with two interval classes separated by a threshold which are given names which are made to imply quality. It may be the case that the employed software metrics insufficiently characterize the two classes to be identified. A set including additional product metrics may improve the model. Pooling product, process, and resource metrics may lead to a better characterization and a model with greater predictive accuracy. Using number of code lines added or deleted from a module may produce more accurate data than the number of faults found. These are important areas for future investigation.

The evolutionary neural network approach appears promising for a broad range of software engineering problems where historical data are available. With the accumulating mass of information on past software development projects, it becomes possible to develop evolutionary data mining techniques to draw latent lessons from it all. Although genetic algorithms and other evolutionary computation methods have been little used in most software engineering domains, the rapidly growing records of success in other fields should be noted for the examples they provide.

Future work may include comparison of evolutionary neural networks to other machine-learning classification approaches. This classification technique could also be compared to a prediction technique combined with a simple decision rule. One would also do well to compare any machine-learning approach with human parameter choices. Finally, future work may also include validation of the practical usefulness of the approach in an industrial software development setting.

ACKNOWLEDGMENTS

We thank Stephen J. Aud and Jean Mayrand for collecting the data used in this research. We are indebted to Donald R. Tveter for the original form of the neural net code which became the kernel of our program. We also thank the anonymous reviewers for their helpful comments and suggestions. This work was supported in part by a grant from Nortel Networks to Florida Atlantic University. The findings and opinions in this paper belong solely to the authors, and are not necessarily those of the sponsor. Moreover, our results do not in any way reflect on the quality of the sponsor's software products.

REFERENCES

[1] N. Baba, H. Handa, and M. Hayashi. Utilization of neural networks and GAs for constructing an intelligent decision support system to deal stocks. In S. K. Rogers and D. W. Ruck, editors, *Applications and Science of Artificial Neural Networks II*, volume 2760 of *Proceedings of SPIE*, pages 164-174, Orlando, FL, Apr. 1996. SPIE–International Society for Optical Engineering.

[2] Bell Canada. *Datrix Metric Reference Manual*. Montreal, Quebec, Canada, version 4.0 edition, May 2000. For Datrix version 3.6.9.

[3] A. L. Blum, and R. L. Rivest. Training a 3-node neural network is NP-complete. *Neural Networks* 5:117-127, 1992.

[4] D. J. Chalmers. The evolution of learning: An experiment in genetic connectionism. In D. S. Touretsky, J. L. Elman, T. J. Sejnowski, and G. E. Hinton, eds., *Proceedings of the 1990 Connectionist Models Summer School*, Morgan Kaufmann, 1990.

[5] L. Fausett. *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, New Jersy, 1994.

[6] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, 1997.

[7] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Massachusetts, 1989.

[8] S. A. Harp, T. Samad, and A. Guha. Towards the genetic synthesis of neural networks. In *Proceedings of the Third International Conference on Genetic Algorithms* (ICGA-89) San Mateo, California, pages 360-369. Morgan Kaufmann, 1989.

[9] R. Hecht-Nielsen. *Neurocomputing*. Addison-Wesley, Reading, Massachusetts, 1991.

[10] J. Hertz, A. Krogh, and R. G. Palmer. *Introduction to the Theory of Neural Computation*. Addison-Wesley, Reading, Massachusetts, 1991.

[11] R. Hochman, T. M. Khoshgoftaar, E. B. Allen, and J. P. Hudepohl. Using the genetic algorithm to build optimal neural networks for fault-prone module detection. In Proceedings: *The Seventh International Symposium on Software Reliability Engineering*, White Plains, New York, pages 152-162, IEEE Computer Society, Oct. 1996.

[12] R. Hochman, T. M. Khoshgoftaar, E. B. Allen, and J. P. Hudepohl. Evolutionary neural networks: A robust approach to software reliability. for fault-prone module detection. In *Proceedings: The Eighth International Symposium on Software Reliability Engineering*, Albuquerque, New Mexico, pages 13-26, IEEE Computer Society, Nov. 1997.

[13] J. H. Holland. *Adaptation in Natural and Artificial Systems* University of Michigan Press, Ann Arbor, Michigan, 1975.

[14] T. M. Khoshgoftaar and E. B. Allen. Neural networks for software quality prediction. In W. Pedrycz and J. F. Peters, editors, *Computational Intelligence in Software Engineering*, volume 16 of *Advances in Fuzzy Systems – Applications and Theory*, pages 33-63. World Scientific, Singapore, 1998.

[15] T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, S. J. Aud, and J. Mayrand. Selecting software metrics for a large telecommunications system. In *Proceedings of the Fourth Software Engineering Research Forum*, pages 221-229, Boca Raton, Florida, Nov. 1995.

[16] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13(1):65-71, January 1996.

[17] T. M. Khoshgoftaar and D. L. Lanning. A neural network approach for early detection of program modules having high risk in the maintenance phase. *Journal of Systems and Software*, 29(1):85-91, April 1995.

[18] T. M. Khoshgoftaar, D. L. Lanning, and A. S. Pandya. A comparative study of pattern recognition techniques for quality evaluation of telecommunications software. *IEEE Journal on Selected Areas in Communications*, 12(2):279-291, February 1994.

[19] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems* 4:461-476, 1990.

[20] D. L. Lanning and T. M. Khoshgoftaar. The impact of software enhancement on software reliability. *IEEE Transactions on Reliability* 44(4):677-682, Dec. 1995.

[21] J.-H. Lin and J. S. Vitter. Complexity results on learning by neural nets. *Machine Learning* 6:211-230, 1991.

[22] B. F. J. Manly. *Multivariate Statistical Methods: a Primer*, 2nd ed. Chapman & Hall, London, 1994.

[23] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308-320, Dec. 1976.

[24] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd ed. Springer-Verlag, Berlin, 1996.

[25] G. F. Miller, P. M. Todd, and S. U. Hegde. Designing neural networks using genetic algorithms. In J. D. Schaffer, ed., *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, California. Morgan Kaufmann, 1989.

[26] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, Massachusetts, 1996.

[27] D. J. Montana and L. D. Davis. Training feedforward networks using genetic algorithms. In *Proceedings: 11th International Joint Conference on Artificial Intelligence*. Palo Alto, California, Morgan Kaufmann, 1989.

[28] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, 5th edition, 2001.

[29] SAS Institute. *SAS/STAT User's Guide*. vol. 1, version 6, 4th ed., Carey, North Carolina, Feb. 1990.

[30] J. D. Schaffer, R. A. Caruana, and L. J. Eshelman. Using genetic search to exploit the emergent behavior of neural networks. In S. Forrest (Ed.), *Emergent Computation: Self-Organizing, Collective, and Cooperative Phenomena in Natural and Artificial Computing Networks*, pages 244-248. MIT Press, Cambridge, Massachusetts, 1991.

[31] G. A. F. Seber. *Multivariate Observations*. John Wiley, New York, 1984.

[32] X. Yao. A review of evolutionary artificial neural networks. *International Journal of Neural Systems* 8(4):539-567, 1993.

[33] J. H. Zar. *Biostatistical Analysis*, 2nd ed. Prentice Hall, Englewood Cliffs, New Jersy, 1984.

[34] H. Zuse. *Software Complexity: Measures and Methods*. deGuryter, Berlin, 1991.

A Fuzzy Model and the *AdeQuaS* Fuzzy Tool: a theoretical and a practical view of the Software Quality Evaluation

Kelly R. Oliveira[1] and Arnaldo D. Belchior[2]

Mestrado em Informática Aplicada
Universidade de Fortaleza
Av. Washington Soares 1321, Fortaleza, CE 60.811-341, Brazil
[1]kellyr@tutopia.com.br
[2]belchior@unifor.br

ABSTRACT

This work introduces the Fuzzy Model for Software Quality Evaluation and its implementation the AdeQuaS Fuzzy tool. The model proposed here comprises a five-stage evaluation process, and it may involve three distinct situations. In the first situation, the evaluation objective is to establish a quality standard for the software product or application domain in question. In the second one, the quality evaluation of a software product is executed, based upon a pre-defined quality standard. In the third, a quality estimation of a software product is found when there is not quality standard available. The AdeQuaS Fuzzy tool, which is based on the Fuzzy Model, has the objective of supporting the stages of software evaluation process, in order to get more effective results about the quality degree of subjective attributes through the judgment of a group of specialists. Besides, it is presented two applications. The first is the evaluation process to e-commerce websites quality. The second is an evaluation of software requirements specifications quality.

KEYWORDS

Software quality evaluation, Software Quality Measures, Quality Evaluation Model, Quality Evaluation Tool, Fuzzy Theory.

1. INTRODUCTION

Software evaluation structures aim primarily to estimate software quality using basic attributes set to underline its main features. The information about the object under analysis must be arranged orderly, so that specific software characteristics can be readily identified to optimize the decision making process [Boloix 1995]. The decision-making process can be seen as the selection of alternatives that are “good enough”, or the choice of action courses so as to attain a certain goal. This process involves uncertainties and, thus, it is necessary that we have the ability to handle imprecise, vague information, taking into account different views, attitudes and beliefs of the involved parts [Ribeiro 1996]. Therefore, it is important to establish the connection between the information and the decision made by a certain individual, who is expected to choose one of many possible actions, in different areas [Simonelli 1996].

Since the decision-making process is centered in human beings, as well as the software evaluation in itself, we find inherent subjectivities and inconsistencies in the problem definition. Thus, fuzzy sets are potentially adequate to deal with this problem, since [Ibrahim and Ayyub 1992]: *(i)* they are capable of representing attributes, *(ii)* they have convenient ways to combine the attributes that can be defined, vaguely or precisely, and *(iii)* they can handle different degrees of importance for each attribute considered.

Just like many others areas of human knowledge, the software quality evaluation involves the appreciation of multiples attributes, as judged by a group of specialists. Each specialist has his or her own opinion and estimates a rank for each attribute, according to either to his or her perception or to the depth of his or her understanding of the problem. Thus, there is a great interest on obtaining an aggregation process, which can consolidate the consensus among the specialists involved in the analysis.

In the process of software quality evaluation, it is not sufficient to identify the attributes that are determinant for the quality. It is also important to determine the procedures that one must follow to control the development process and attain the desired quality level. This is accomplished through the application of some metrics, in an organized and well-planned way, which makes the developers more conscious of both the relevance of management and the commitment with a quality standard.

A model for quality evaluation must support the use of software quality metrics [Fenton and Pfleeger 1997; Kitchenham *et al.* 1996; Möller 1993, Schneidewind 1992], so that we can conveniently achieve our goals. In this work, we will present the Fuzzy Model for Software Quality Evaluation (FMSQE) [Belchior 1997] to evaluate the software quality, because it has already been satisfactorily and efficiently used in several applications [Albuquerque 2001; Branco Jr. and Belchior 2001; Campos *et al.* 1998]. Its use will be done through the *AdeQuaS* Fuzzy tool, which adds much functionality in an evaluation process.

This paper is organized in five sections. Section 2 gives the general vision of the process of software quality evaluation, focusing the quality model and metrics use. The section 3 presents how the Fuzzy Theory can be used in software quality. Moreover, the FMSQE is described through its stages. The fourth section presents the *ADEQUAS* tool. In addition, two evaluation results are showed: e-commerce websites quality and software requirements specifications. Finally, in section 5, the conclusions are drawn.

2. SOFTWARE QUALITY EVALUATION

The quality requirements specification is one of the arduous stages of an evaluation process [Boehm and Hoh 1996; Rocha *et al.* 2001]. Its result is the model quality that will be used in the evaluation. The quality model corresponds to a relevant attributes set of a certain product that has to be adequately adapted to the evaluation context [Pfleeger 1998; Pressman 2000].

A general quality model is proposed by ISO/IEC 9126-1 [ISO 2001], which can be applied to any kind of software. It is divided in two parts: *(i)* internal quality and external quality, and *(ii)* quality in use.

For internal quality and external quality, characteristics and subcharacteristics are described, related to internal operation and its behavior into an external environment. For quality in use, characteristics are described, related to operational context in the user's point of view.

The characteristics and subcharacteristics have to be observed under certain conditions, according to the evaluation purpose. They are:

- Functionality: refers to the presence of functions that satisfy the user's needs. Includes: suitability, accuracy, interoperability, security and functionality compliance;
- Reliability: corresponds to maintenance of appropriated performance. Includes: maturity, fault tolerance, recoverability and reliability compliance;
- Usability: refers to using, understanding and learning facility. Includes: understandability, learnability, operability, attractiveness and usability compliance;
- Efficiency: means a well use of resources with maintenance of performance. Includes: time behavior, resource utilization and efficiency compliance;
- Maintainability: refers to the modification and correction facility. Includes: analyzability, changeability, stability, testability and maintainability compliance;

- Portability: means the environment change adaptability. Includes: adaptability, installability, co-existence, replaceability and portability compliance.

The evaluation process presents the stages that must be followed by the participants. The ISO/IEC 14598 [ISO 1998] defines the process in four stages:

- Establishing the evaluation requirements;
- Specifying evaluation;
- Projecting evaluation;
- Executing evaluation.

On the first step, it is necessary to establish the evaluation requirements, so that the evaluation goals, the object to be evaluated, and the quality model will be identified.

The next step is to specify the evaluation through metrics definition and punctuation, as well as its judgment. Each metric has to be carefully quantified and related to a quality characteristic. The punctuation must be mapped in a satisfaction scale, that indicates if the software is within the stated limits between what is acceptable or not.

Next, projecting the evaluation consists of planning the procedures to be executed by the evaluator, including action methods and time schedule.

By executing the evaluation, the metrics related to the object have to be collected and, subsequently, compared with a predetermined satisfactory punctuation.

The Fuzzy Model stands out from other quality models because:

- It includes all the stages of evaluation process, from the definition until the presentation of results, according to ISO 14598 [ISO 1998];
- It gets the importance degree of the judgment of each specialist, according to his or her professional and academic profile;
- It shows, in a quantitative way, the imprecise concepts and qualitative attributes;
- It gets a quality index, which indicates the quality of the software.

In next section, we will give a brief overview of how to Fuzzy Theory can be applied in software quality evaluations, so as to provide a better understanding of our work. In addition, the Fuzzy Model for Software Quality Evaluation is described.

3. THE FUZZY MODEL FOR SOFTWARE QUALITY EVALUATION

The Fuzzy Set Theory has been used in several areas of human knowledge as the link between the imprecise (subjective) models of the real world and their mathematical representations [Araújo 2000; Dubois and Prade 1980]. In Software Engineering, many applications have been developed, such as the following measure techniques: f-COCOMO [Idri *et al.* 2000; Ryder 1998], Function points [Gray 1997; Lima Jr. *et al.* 2001], McCall Method [Pedrycz and Peters 1998] and Delphi Method [Ishikawa *et al.* 1993].

The Fuzzy Logic is an extension of the traditional logic that excludes the dualistic vision of true and false simultaneously exclusive. "In fact, between the sure of to be and the sure of not to be, there are infinite unsure degrees" [Sousa 1995]. Starting from this presupposition, between the true (1) and false (0) are considered infinite values in the interval [0,1] that indicates the true degree (or membership degree) of a certain set element. For instance, it can be considered a tones scale of gray between the black and the white.

A fuzzy set is characterized by a membership function, which maps the elements of a domain, space or discourse universe X for a real number in [0, 1]. Formally, $\tilde{A} : X \rightarrow [0, 1]$. Thus, a fuzzy set is presented as a set of ordered pairs in which the first element is $x \in X$, and the second, $\mu_{\tilde{A}}(x)$, is the degree of membership or the membership function of x in $\tilde{A}$, which maps x in the interval [0, 1], or, $\tilde{A} = \{(x, \mu_{\tilde{A}}(x)) \mid x \in X\}$ [20]. The membership of an element within a certain set becomes a question of degree, substituting the actual dichotomic process imposed by set theory, when this treatment is not suitable. In extreme cases, the degree of membership is 0, in which case the element is not a member of the set, or the degree of membership is 1, if the element is a 100% member of the set [20].

In this theory's view, each quality attribute can be seen as a linguistic variable, related with a set of linguistic terms, associated with membership functions, in a reference set previously established. Each quality attribute will be a composition of linguistic terms, obtained in an evaluation process.

The linguistic terms Ti, for $i = 1, 2, \ldots, n$, will be represented by LR-type normal triangular fuzzy numbers $\tilde{N}i$ (a_i, m_i, b_i) [Bardossy *et al.* 1993; Dubois and Prade 1980, 1991; Hapke *et al.* 1994; Hsu and Chen 1996; Lasek 1992; Lee 1996a, b; Römer and Kandel 1995; Ruoning and Xiaoyan 1992], which denote the importance degree of each attribute considered. It is important to remark that $a_i < b_i$ and $a_i \leq m_i$ or $m_i \leq b_i$. The a and b values identify, respectively, the inferior and superior limits of the triangle base.

Their membership degree is equals 0 ($\mu_{\tilde{A}}(a) = 0$ and $\mu_{\tilde{A}}(b) = 0$). The value of *m* corresponds to the triangle height, where $\mu_{\tilde{A}}(m) = 1$ [Zadeh 1988]. The value of *n* can be set to meet the requirements of the project, the peculiar features of the application domain, or the quality management staff's requirements.

In the similar way, a trapezoidal fuzzy number can be represented by $\tilde{N}$ (*a*, *m*, *n*, *b*). The *a* and *b* values identify the lower and upper limits of the larger base of the trapezoid. The *m* and *n* values are, respectively, the lower and upper limits of the smaller base of the trapezoid [Dubois and Prade 1991].

Each fuzzy number represents an importance degree and uses a linguistic term as a meaning. For such point of view, based on [Baldwin 1979; Hsu and Chen 1996; Kacprzyket *et al.* 1992; Lee 1996a, b; Palermo and Rocha 1989], in Table 1 is shown an example of a linguistic terms scale with four values, which can be used in the quality evaluation of a software product. Each value can be transformed to a normal triangular fuzzy number using fuzzification. This scale will be used in this paper as example of fuzzy model and evaluation tool application, described shortly afterwards.

The set of linguistic terms shown above has the following membership functions represented in Figure 1, and adapted from Lee [Lee 1996a, b].

Scale	Fuzzy Number	Linguistic Term
0	Ñ = (0.0, 0.0, 1.0)	No relevance
1	Ñ = (0.0, 1.0, 2.0)	Slightly relevant
2	Ñ = (1.0, 2.0, 3.0)	Relevant
3	Ñ = (2.0, 3.0, 4.0)	Very relevant
4	Ñ = (3.0, 4.0, 4.0)	Indispensable

Table 1. Scale Example for Quality Attribute Evaluation using Normal Fuzzy Numbers.

The FMSQE inherits the fuzzy theory robustness. It actuates as a mechanism that is able to interpret results and summarize information through ruled procedures to quality evaluation.

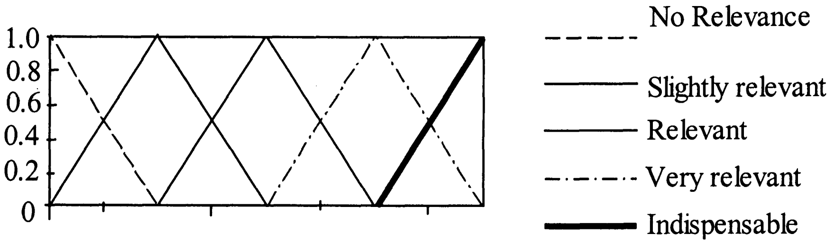


Figure 1. Membership functions for linguistic terms [Belchior 1997].

This model involves many stages of an evaluation process, since the quality model determination until the assessment execution [ISO 1998]. Besides, it allows getting the consensus degree of a group of specialists, considering the importance degree of the judgment, based on experience levels of each one.

Attaining its objectives, the FMSQE, which is described in details in [Belchior 1997], is defined in five steps. Such objectives may involve three distinct situations:

- *Quality Standard (QS) Determination for software product or application domain*: specialists on a certain product (or an application domain) determine the importance degree of each attribute, in order to get a satisfactory quality level of the product. It means that the weight assigned to each attribute by a specialist has to portrayal how the product ought to be. Thus, in this case, we are not evaluating a certain product state, but the ideal quality standard that it should present. In this context, the "quality standard" can be understood as a guide to quality evaluation in an specific application domain.
- *Evaluation of a software product quality, based on a predefined QS*: each specialist judges the quality attributes set, considering the software state. The outcome of such appraisal is compared with the specific predefined QS to the product or the application domain that is being evaluated. A quality index for each considered attribute is generated, and thus the measurement of the final product quality is performed. The indexes mean the percentage that the product attains according to the quality standard.
- *Evaluation of a software product, without predefined QS*: the results will be investigated, taking into account only the specialists' appraisal. This procedure generates a set of useful data that can be used by the development staff or by the product quality manager. These data can help them to carry on the product development or serve as a parameter to improve a final product.

The FMSQE, presented in Figure 2, extends the Rocha Model [Rocha 1983] by the fuzzy theory application in order to facilitate the quantification of qualitative concepts. The main concepts are:

- *Quality characteristics*: is an attribute set of a software product that allows describing and evaluating this product. A software quality characteristic can be detailed in multiple levels of subcharacteristics. The lowest subcharacteristic level (metric) is called primitive subcharacteristic and it is susceptible to evaluation.
- *Evaluation processes*: determines the process and the instruments to be used, in order to measure the presence degree of a specific metric;
- *Metrics*: are the product evaluation results, according to primitive subcharacteristics, through fuzzy linguistic terms, mapped by fuzzy numbers;
- *Aggregated measurements*: are the metric aggregation results, gotten from the evaluation of the primitive subcharacteristics. They are also primitive subcharacteristics aggregation results into subcharacteristics or characteristics and into the final quality value of the software product;
- *Fuzzy* functions: maps the primitive or aggregated quality attributes, through linguistic terms, quantifying them.

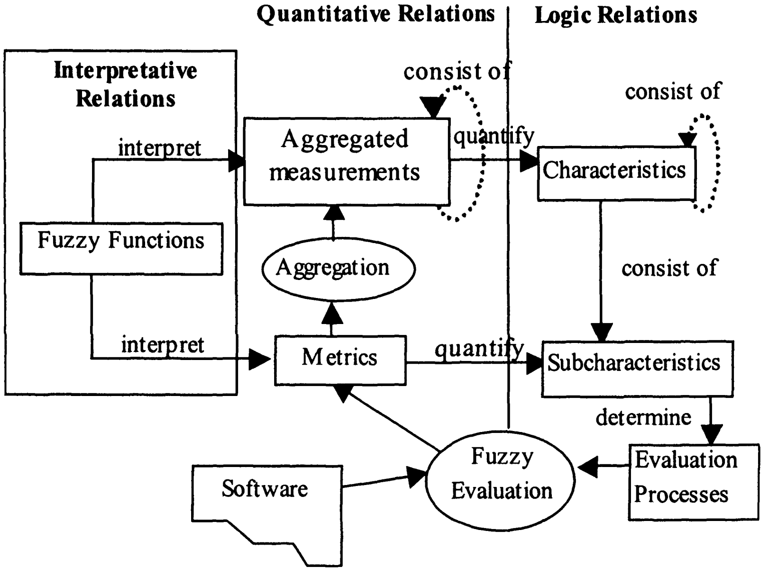


Figure 2. Fuzzy Model for Software Quality Evaluation.

The stages of FMSQE are the following:

First Stage

It consists of the identification of the object to be evaluated, the considered quality attributes set and the institutions that the product will be tested.

- *Establishing the object to be evaluated:* in this stage, it should be determined which software product will be evaluated. It can be evaluated intermediary products or the final product
- *Defining the quality attributes set*: The set of attributes is defined according to the evaluation's object, the application domain and the development technology. Its consists of an hierarchical tree of software quality attributes. Two situations may occur. In the first one, the attributes have already been defined in previous works. There are several attributes sets based on scientific [Kacprzyk *et al.* 1992], financial, educational, medical, specialists systems, information systems and object-oriented software. In the second one, the attributes have not been defined yet, and it must be proceeded their identification. This task can be accomplished through the use of the tool currently in development;
- *Selecting the institution(s) that will support the research:* can be *(i)* just one institution: when an institution performs its own quality attributes evaluation of a software product, or *(ii)* several institutions: when the data is collected in order to be defined the quality standard of a certain software product, in a specific application domain.

Example:

- Evaluation object: Software Requirements Specifications (SRS)
- Set of attributes: Clunie [1997] defined the hierarchical tree of software quality attributes that has three levels: objectives, factors, and subfactors. There are three objectives distributed in the follow:.
- Representation reliability: 2 factors and 8 subfactors;
- Conceptual reliability: 2 factors and 5 subfactors;
- Utilizability: 4 factors and 12 subfactors.
- *Institutions where the survey was applied*: 2 companies that has large experience in SRS elaboration, and 1 university.

Second Stage

It is the choice of the all the specialists who will participate in the evaluation process.

- *Defining the specialists' profile:* in this stage, it will be gotten the specialists' profile [Fenton and Pfleeger 1997], *Ei* (*i* = 1, 2, …, *n*). The specialists will participate of the investigation process, through the

Specialist Profile Identification Questionnaire (SPIQ), in order to point out the relative importance of each one. Generally speaking, every person, either direct or indirectly, who is or has been involved with products that are similar to the object under analysis, can be chosen as a specialist. The SPIQ is constituted of *n* questions (i_1, i_2, ..., *in*) which objective is to evaluate each specialist, involving essentially his or her experience in system development and his or her training in computer science;

- *Determining the specialist's weight*: it should be calculated the relative importance degree of each specialist, by the weight w_{Ei} generation through SPIQ data and taking into account the following criteria: *(i)* each SPIQ contains information about just one specialist, *(ii)* the total score of each specialist, $tSPIQ_i$, is calculated according to the signs contained in the SPIQ results [Belchior 1997], and *(iii)* the weight of each specialist w_{Ei}, which is the his or her relative weight in relation to the other specialists (weighted mean), is defined as:

$$w_{Ei} = \frac{tSPIQi}{\sum_{i=1}^{n} tSPIQi}$$

Example:

First, specialists with experience in SRS are selected. Each specialist answers the SPIQ with seven questions. The profile of the specialists, *Ei*, was calculated through the SPIQ punctuation. Next, the relative weight of each specialist is calculated in relation of the sum of punctuation of the others specialists.

Third Stage

It consists of the rank determination of each quality attribute, identified in the First Stage.

The investigation consists of obtaining from the selected specialists the rank of each attribute, in order to get the appraisal of each one relatively to each measurable quality attribute, through the set of linguistic terms characterized by fuzzy numbers *Ñi* (a_i, m_i, b_i), previously designed.

At this stage, the specialist must have been informed that the investigation process is in progress either to determine a certain Quality Standard (QS), or just to evaluate the state of art of a particular software product. This information is necessary so that the specialist can make his or her appraisal coherently. This stage comprises the following activities:

- *Defining of the investigation procedure*: this procedure can consist of the assembling of a questionnaire or any other investigation device, and in the definition of the suitable application techniques, using importance degrees (of the linguistic terms) previously set;
- *Applying the investigation device*: the specialists, who were selected in previous stage, evaluate through the investigation device.

Example:

In this stage, the assessment questionnaire (AQ), which consists of a list of relevant attributes on SRS with their , was elaborated. The AQ consist of a list of relevant attributes in SRS (obtained in Stage 1) and their possible answers (linguistic terms associated to fuzzy numbers). After elaboration, the AQ is distributed among the specialists.

Fourth Stage

It is the moment that occurs the treatment of the data provided by the specialists in the evaluation of each measurable quality attribute considered (metric).

The individual prognoses from each specialist for the directly measurable quality attributes (metrics) are combined, generating a consensus of the specialists, for each metric evaluated. This consensus is formally expressed through a characteristic membership fuzzy function $\tilde{N}$ [Hsu and Chen 1996]:

$$\tilde{N} = f(\tilde{N}_1, \tilde{N}_2, \ldots, \tilde{N}_n)$$

- *Calculating the agreement degree:* the agreement degree is calculated [Hsu and Chen 1996, Chen and Hsy 1993], $A(\tilde{N}_i, \tilde{N}_j)$, combining the appraisals of the specialists, E_i and E_j, through the ratio of the intersection area to the total area of their membership functions:

$$A(\tilde{N}_i, \tilde{N}_j) = \frac{\int (\min\{\mu_{\tilde{N}_i}(x), \mu_{\tilde{N}_j}(x)\})dx}{\int (\max\{\mu_{\tilde{N}_i}(x), \mu_{\tilde{N}_j}(x)\})dx}$$

- *Assembling the agreement matrix*: after having calculated all the agreement degrees between each pair of specialists E_i and E_j, an agreement matrix should be assembled [Zwick *et al.* 1987], *AM*, which indicates the consensus among the specialists:

$$AM = \begin{bmatrix} 1 & A_{12} & \dots & A_{1j} & \dots & A_{1n} \\ \mathrm{M} & \mathrm{M} & & \mathrm{M} & \mathrm{M} & \mathrm{M} \\ A_{i1} & A_{i2} & \dots & A_{ij} & \dots & A_{in} \\ \mathrm{M} & \mathrm{M} & & \mathrm{M} & \mathrm{M} & \mathrm{M} \\ A_{n1} & A_{n2} & \dots & A_{nj} & \dots & 1 \end{bmatrix}$$

Once the matrix is assembled, it must be observed the following points:

1. If Cij = 0, so there is not intersection between the ith and the jth specialist. In this case, it is necessary get more information from these specialists (according to the evaluation's convenience), in order to make their opinions converge to a point or, in other words, to find an intersection between them.

2. After having collected the additional information referred to item (i), if any concordance degree is still zero, it is taken into matrix AM anyway, because, in the process of aggregation, the values that are equal to zero (those which point out to a disagreement among the specialists) will be assigned zero weight.

3. It must be paid attention to cases that there is a great disparity among the answers (low degree of agreement among the specialists), because it can mean that they did not understand conveniently the definition of the object of investigation [Dyer 1992]. In this case, item (i) must be redone, as many times as necessary to reach a higher consensus among the specialists.

- *Calculating the relative agreement:* using the data given by *AM*, it is calculated the relative agreement (*RA*i) of each specialist involved in the process, through the root quadratic mean of the agreement degree among them:

$$RAi = \sqrt{\frac{1}{n-1}\sum_{\substack{j=1 \\ j \neq i}}^{n} A_{ij}^{\ 2}}$$

This procedure assures that the *RAi* determination will tend to higher indexes of consensus among the specialists in charge of the evaluation;

- Calculating *the degree of relative agreement: t*he relative agreement degree (*RAD*i) of a specialist, relatively to all the others, is obtained through the weighted mean of *RA*i from each specialist:

$$RADi = \frac{RA_i}{\sum_{i=1}^{n} RA_i}$$

- *Calculating the specialists' consensus coefficient:* the consensus coefficient, obtained for each specialist (*SCC*i) will take into consideration both the *RAD*i and the weight w_{Ei}, of each specialist [Hsu and Chen 1996]:

$$\mathrm{SCC_i} = \frac{RAD_i \cdot w_{Si}}{\sum\limits_{i=1}^{n}(RAD_i \cdot w_{Si})}$$

- *Evaluating the metrics quality:* the outcome of the evaluation of each metric quality is given by $\tilde{N}$, that is also a normal triangular fuzzy number, where • is the algebraical fuzzy product [Hsu and Chen 1996; Kaufmann and Gupta 1991]:

$$\tilde{N} = \sum_{i=1}^{n}(SCC_i \bullet \tilde{N}_i)$$

<u>Example</u>:

In Table 2 is shown the fuzzy numbers (the specialists choose the linguistic term associated to these fuzzy numbers) that represent the values of the factor "Conciseness".

Specialists	Fuzzy Number Ñ (a, m, b)
1	(2.00, 3.00, 4.00)
2	(2.00, 3.00, 4.00)
3	(3.00, 4.00, 4.00)
4	(3.00, 4.00, 4.00)
5	(3.00, 4.00, 4.00)
6	(2.00, 3.00, 4.00)
7	(1.00, 2.00, 3.00)
8	(3.00, 4.00, 4.00)
9	(2.00, 3.00, 4.00)
10	(2.00, 3.00, 4.00)
11	(3.00, 4.00, 4.00)
12	(2.00, 3.00, 4.00)
13	(3.00, 4.00, 4.00)
14	(2.00, 3.00, 4.00)
15	(3.00, 4.00, 4.00)
16	(3.00, 4.00, 4.00)

Table 2. Fuzzy numbers that the specialists choose in relation to factor "Conciseness".

Each element of the agreement matrix is calculated through the agreement of all the possible pairs of specialists, E_i e E_j, as shown in Table 3.

The relative agreement of the specialist E_1 is calculated as follow:

$$RA_1 = \sqrt{\frac{1}{16-1}(1.0^2 + 0.2^2 + 0.2^2 + 0.2^2 + 1.0^2 + 1.14^2 + \mathrm{K} + 0.2^2)}$$

The degree of relative agreement of the specialist E_1:

$$RAD_1 = \frac{0.6501}{10.1723} = 0.06391$$

The consensus coefficient of specialist E_1 and of the specialist E_{16}:

$$SCC_1 = \frac{0.06391 \bullet 0.0592}{0.0610} = 0.0621$$

The metrics quality evaluation is equal the follow result:

$$\tilde{N} = \left\{\left[0.0621 \bullet \tilde{N}_1\right] + \mathrm{K} + \left[0.0792 \bullet \tilde{N}_{16}\right]\right\}$$

$$\tilde{N} = \left\{\begin{matrix} \left[(0.0621 \bullet 2.0) + \mathrm{K} + (0.0792 \bullet 3.0)\right]; \\ \left[(0.0621 \bullet 3.0) + \mathrm{K} + (0.0792 \bullet 4.0)\right]; \\ \left[(0.0621 \bullet 4.0) + \mathrm{K} + (0.0792 \bullet 4.0)\right] \end{matrix}\right\}$$

$$\tilde{N} = (2.55\,; 3.55; 3.99)$$

Agreement Matrix

E_i/E_j	E_1/E_j	E_2/E_j	E_3/E_j	E_4/E_j	E_5/E_j	E_6/E_j	E_7/E_j	E_8/E_j	E_9/E_j	E_{10}/E_j	E_{11}/E_j	E_{12}/E_j	E_{13}/E_j	E_{14}/E_j	E_{15}/E_j	E_{16}/E_j
E_i/E_1	1.0	1.0	0.2	0.2	0.2	1.0	0.14	0.2	1.0	1.0	0.2	1.0	0.2	1.0	0.2	0.2
E_i/E_2	1.0	1.0	0.2	0.2	0.2	1.0	0.14	0.2	1.0	1.0	0.2	1.0	0.2	1.0	0.2	0.2
E_i/E_3	0.2	0.2	1.0	1.0	1.0	0.2	0.0	1.0	0.2	0.2	1.0	0.2	1.0	0.2	1.0	1.0
E_i/E_4	0.2	0.2	1.0	1.0	1.0	0.2	0.0	1.0	0.2	0.2	1.0	0.2	1.0	0.2	1.0	1.0
E_i/E_5	0.2	0.2	1.0	1.0	0.2	0.2	0.14	1.0	0.2	0.2	1.0	0.2	1.0	0.2	1.0	0.2
E_i/E_6	1.0	1.0	0.2	0.2	1.0	1.0	0.0	0.2	1.0	1.0	0.2	1.0	0.2	1.0	0.2	1.0
E_i/E_7	0.14	0.14	0.0	0.0	0.0	0.14	1.0	0.0	0.14	0.14	0.0	0.14	0.0	0.14	0.0	0.0
E_i/E_8	0.2	0.2	1.0	1.0	1.0	0.2	0.0	1.0	0.2	0.2	1.0	0.2	1.0	0.2	1.0	1.0
E_i/E_9	1.0	1.0	0.2	0.2	0.2	1.0	0.14	0.2	1.0	1.0	0.2	1.0	0.2	1.0	0.2	0.2
E_i/E_{10}	1.0	1.0	0.2	0.2	0.2	1.0	0.14	0.2	1.0	1.0	0.2	1.0	0.2	1.0	0.2	0.2
E_i/E_{11}	0.2	0.2	1.0	1.0	1.0	0.2	0.0	1.0	0.2	0.2	1.0	0.2	1.0	0.2	1.0	1.0
E_i/E_{12}	1.0	1.0	0.2	0.2	0.2	1.0	0.14	0.2	1.0	1.0	0.2	1.0	0.2	1.0	0.2	0.2
E_i/E_{13}	0.2	0.2	1.0	1.0	1.0	0.2	0.0	1.0	0.2	0.2	1.0	0.2	1.0	0.2	1.0	1.0
E_i/E_{14}	1.0	1.0	0.2	0.2	0.2	1.0	0.14	0.2	1.0	1.0	0.2	1.0	0.2	1.0	0.2	0.2
E_i/E_{15}	0.2	0.2	1.0	1.0	1.0	0.2	0.0	1.0	0.2	0.2	1.0	0.2	1.0	0.2	1.0	1.0
E_i/E_{16}	0.2	0.2	1.0	1.0	1.0	0.2	0.0	1.0	0.2	0.2	1.0	0.2	1.0	0.2	1.0	1.0

Table 3. Matrix of agreement between the specialists E_i e E_j. in relation to factor "Conciseness".

Thus, the fuzzy number that represent the factor Conciseness is $\tilde{N}$=(2.55, 3.55, 3.99). The same procedure is executed to all others quality attributes.

Fifth Stage

It consists of the aggregation of the software quality attributes, at each hierarchical level of the quality model.

At this stage, it is accomplished the aggregation of the quality attributes of type $\tilde{N}$, generating a characteristic membership fuzzy function for each subset of quality attributes that means the aggregated ones. Each aggregated attribute evaluated, $\tilde{N}$, composed of its constituent attributes, $\tilde{N}_{C1}$, $\tilde{N}_{C2}$, …, $\tilde{N}_{Cn}$, will be formally represented as:

$$\tilde{N} = f(\tilde{N}_{C1}, \tilde{N}_{C2}, \dots, \tilde{N}_{Cn})$$

- *Establishing the quality standard*: the establishment of the quality standard (QS) for a certain software product or a specific application domain requires the calculation of the weight W_i [Belchior 1997; Liou *et al.* 1992]. It means the contribution degree of each attribute comprising the aggregated attribute evaluated. The weight of each attribute, W_i, is obtained through the calculation of the weighted mean of the importance degrees of each constituent attribute, *wi*, which is calculated by defuzzying its correspondent fuzzy number $\tilde{N}$i (a_i, m_i, b_i). Therefore:

  i. $w_i = m_i$, which corresponds to a membership degree whose value is equal to 1. This value means the crisp number of the quality attribute.

  ii. $W_i = w_i / \Sigma w_i$

- *Calculating the degree of aggregated agreement*: we calculate the agreement degree, A ($\tilde{N}$i, $\tilde{N}$j) of the quality attributes that are being aggregated (which are fuzzy numbers $\tilde{N}$);
- *Assembling the aggregation agreement matrix:* once all the C_{ij} of the attributes of the subset that is being aggregated have been calculated, the aggregation agreement matrix (*AAM*) is generated. If $C_{ij} = 0$, so there is not intersection between the attributes *i* and *j*, and, in this case, the degree on disagreement is calculated, C*ij*, among these attributes. This value must be contained in the interval [-1,0]:

$$\overline{C}_{ij} = -\frac{d}{D} \bullet r$$

where:

1. *d* is the shortest distance between two fuzzy numbers considered, that is, $d = a_j - b_i$ or $d = a_i - b_j$ (the smaller absolute value of *d*).

2. *D* is the greater distance between the higher and the lower linguistic term in the set of linguistic terms considered, $\tilde{N}_n$ and $\tilde{N}_1$, respectively. Therefore, $D = a_n - b_1$.

3. The ratio between the areas of the fuzzy numbers $\tilde{N}i$, and $\tilde{N}j$ is *r*, where $0 < r \leq 1$. Thus

$$r = \frac{\int_x (\mu_{\tilde{N}i}(x))dx}{\int_x (\mu_{\tilde{N}j}(x))dx} \quad or \quad r = \frac{\int_x (\mu_{\tilde{N}j}(x))dx}{\int_x (\mu_{\tilde{N}i}(x))dx}$$

- *Assembling the aggregation matrix*: the new aggregation matrix is assembled, which comprises the values of *Cij* and $\bar{C}ij$, that are the degrees of the aggregation states, *Eij*, replacing the matrix *AAM*;
- *Calculating the relative state of aggregation*: the relative state of aggregation (*RSA*) is obtained, examining the agreement or disagreement of each attribute that are being aggregated, through the use of two procedures:

When there are any disagreement degree ($\bar{C}ij$) in the *AM*, it should initially be calculated the "*rsa*" for each attribute, through the weighted mean of the agreement degree and the disagreement degree of that attribute, along with the others that are being aggregated. Thus, it can be quantified the extent that the attribute agrees and disagrees relatively to the others, on the process of aggregation.

$$rsa_i = \frac{1}{n-1} \sum_{\substack{j=1 \\ j \neq i}}^{n} E_{ij}$$

This procedure allows us to view the state of each attribute in the aggregation process. It means how much each attribute contributes in the composition of the new aggregated attribute. When the value of the "*rsa*" of a certain aggregating attribute is non-positive, we can argue whether this attribute should or should not belong to that branch of the attribute composition hierarchy of that particular software product. This can be useful information for the validation of the hierarchical tree of software quality attributes.

Since the attributes that are being aggregated belong, in fact, to the hierarchical branch of the quality attribute tree in question, we proceed to the calculation of the RSA, evaluating the quadratic mean of the aggregation states degrees, Eij, of their aggregating attributes:

$$RSAi = \sqrt{\frac{1}{n} \sum_{j=1}^{n} E_{ij}^{\,2}}$$

In this case, the negative degrees (disagreement) are treated and they influences in the same way as the positive ones. This procedure intends to produce a compensatory value for each attribute's contribution in the process of aggregation, through the Fuzzy Model proposed. Thus, preserving the particular composition characteristics of the attributes that means their location in the hierarchical tree of quality software attributes, established by the organization's quality management staff, it should be executed the process of aggregation.

For example, if a certain software quality attribute comprises two other attributes having different importance degrees, for a certain product and, if the importance degree of one of these attributes is low and the other is high, it means that the percentages of their contributions to the composition of the aggregated attribute considered are, respectively, low and high. Applying procedure (*ii*) above, we can obtain a medium importance degree for the aggregated attribute. Therefore, the aggregated attributed evaluated has a medium importance degree, and comprises two attributes the importance degree of which are low and high.

Nevertheless, if, during the analysis of the results, it is realized that the aggregation process is not be adequate for the product that is being evaluated, we must justify the reason of the aggregation can not be accomplished as such, relating just the importance degrees of each attribute that would be aggregated.

- *Calculating the relative state of aggregation*: the relative state of aggregation degree (*RSAD*) of each aggregated attribute is obtained through the evaluation of the weighted mean of its constituent attributes:

$$RSADi = \frac{RSAi}{\sum_{i=1}^{n} RSAi}$$

- *Calculating the attribute consensus coefficient*: the attribute consensus coefficient (*ACC*i), obtained for each attribute that comprises the one that is being generated, will take into account both the *RSAD* and the weight *Wi* of each attribute. In case it has not still been established a quality standard (*QS*) or it was already determined the QS software for the product that is being evaluated, or for its application domain, we must consider *Wi* = 1, that is, *ACCi* = *GRSDi*:

$$ACCi = \frac{RSADi \bullet Wi}{\sum_{i=1}^{n}(RSADi \bullet Wi)}$$

- *Evaluating the aggregated attribute*: the evaluation result of each aggregated quality attribute is given by $\tilde{N}$, which is also a fuzzy number, where • is the algebraical fuzzy product [Kaufmann and Gupta 1991], formally written as:

$$\tilde{N} = \sum_{i=1}^{n}(ACCi \bullet \tilde{N}i)$$

Based on the results obtained from the application of this fuzzy model in software quality evaluation, we can define quality indexes to guide in the evaluation of new software products, according to the quality standard established.

Example:

This stage is very similar to the previous one. In fourth stage, the result is gotten through the fuzzy numbers that represent the answers of the specialists. In this stage, the result is obtained, initially, through the aggregation of fuzzy numbers calculated in fourth stage, and, after, through the aggregation of fuzzy numbers calculated in each hierarchical level of the tree of quality attributes.

3.1. Definition of software quality indexes

When there is a quality standard defined for the software product, or its application domain, we can compare it with the results obtained in the fuzzy model application. So, we obtain a quality index, that indicates whether the evaluated software product matches the quality standard or not, as well as the percentage of that match.

We can determine the quality index of the software product through the following successive actions:

- *Redefinition of the quality attribute characteristic function:* we redefine the fuzzy function of each quality attribute of triangular type $\tilde{N}_i$ (a_i, m_i, b_i), obtained at the definition of the quality standard, as the quality standard fuzzy function, $\tilde{Q}_i$ (a_i, m_i, $\tilde{m}_i$, b_i) that is a trapezoidal normal fuzzy number.

The mapping of $\tilde{N}_i$ to $\tilde{Q}_i$ will result in the function $\tilde{Q}_i$ (a_i, m_i, b_n, b_n), where n the upper limit of the previously defined referential set. This mapping is possible because we can consider that any value that is higher than the quality standard (situated on the right side of the characteristic function) is also of quality and, consequently, utterly acceptable.

- *Calculation of the quality index:* the quality index, q_k, for each attribute k that is being evaluated, is given by:

$$qk = \frac{\int_X (\min\{\mu\tilde{Q}_i(x), \mu\tilde{N}_k(x)\})dx}{\int_X (\mu\tilde{N}_k(x))dx}$$

Once q ∈ [0.1], when q = 1, it means that the evaluated attribute reaches the quality pattern; if q = 0, then the evaluated element is not up to the standard; if 0 ≤ q ≤ 1, the attribute is found within the limits of the standard, and it reaches q% of the quality standard.

In the calculation of the quality index for aggregated attributes, if any of its constituent attributes has q = 1, we consider the value of QS for this constituent attribute in our calculations.

All this stages are implemented and can be applied using the *AdeQuaS* Fuzzy tool, which is described in the following sections.

4. THE *ADEQUAS* FUZZY TOOL

The *AdeQuaS* Fuzzy tool is an implementation of the FMSQE that has the objective of support its evaluation process stages. In order that, it is comprised of two modules: *AdeQuaS*-Analyzer and *AdeQuaS*-Assessor.

The *AdeQuaS*-Analyzer is the main module, where the most important quality evaluation activities are executed, such as: *(i)* defining evaluation and establishing the purposes, *(ii)* identifying the evaluation object, *(iii)* choosing and recording the specialists, who are the evaluation process participants, *(iv)* elaborating specialists profile identification questionnaire (SPIQ), *(v)* establishing quality requirements, through the assessment questionnaire (AQ) elaboration, and *(vi)* generating reports with the attributes aggregation results.

The module *AdeQuaS*-Assessor has the objective of facilitating the evaluation research, being used by the specialist. It is complementary to and dependent on the main module. For that reason, it consists of a viewer of SPIQ and AQ, which are predefined on module *AdeQuaS*-Analyzer. Besides, the specialist could modify or correct his record, if necessary. The tasks, which can be executed by the evaluator, are: *(i)* viewing the assessment information and the object definition, *(ii)* checking and correcting his own record data, *(iii)* answering the SPIQ, and *(iv)* answering the AQ

4.1. The AdeQuaS Tool

The Figure 3 presents the *AdeQuaS* tool on the evaluation process, focusing in the Analyzer module and the Assessor module.

Initially, the evaluation is defined. The AQ elaboration could be undergone two situations:

- The AQ could be based on a pre-existent quality standard (QS) of an application domain, which the objective is to confront the evaluation results with the quality degrees defined in the chosen QS;

- The AQ could be elaborated based on the definition of relevant quality requirements.

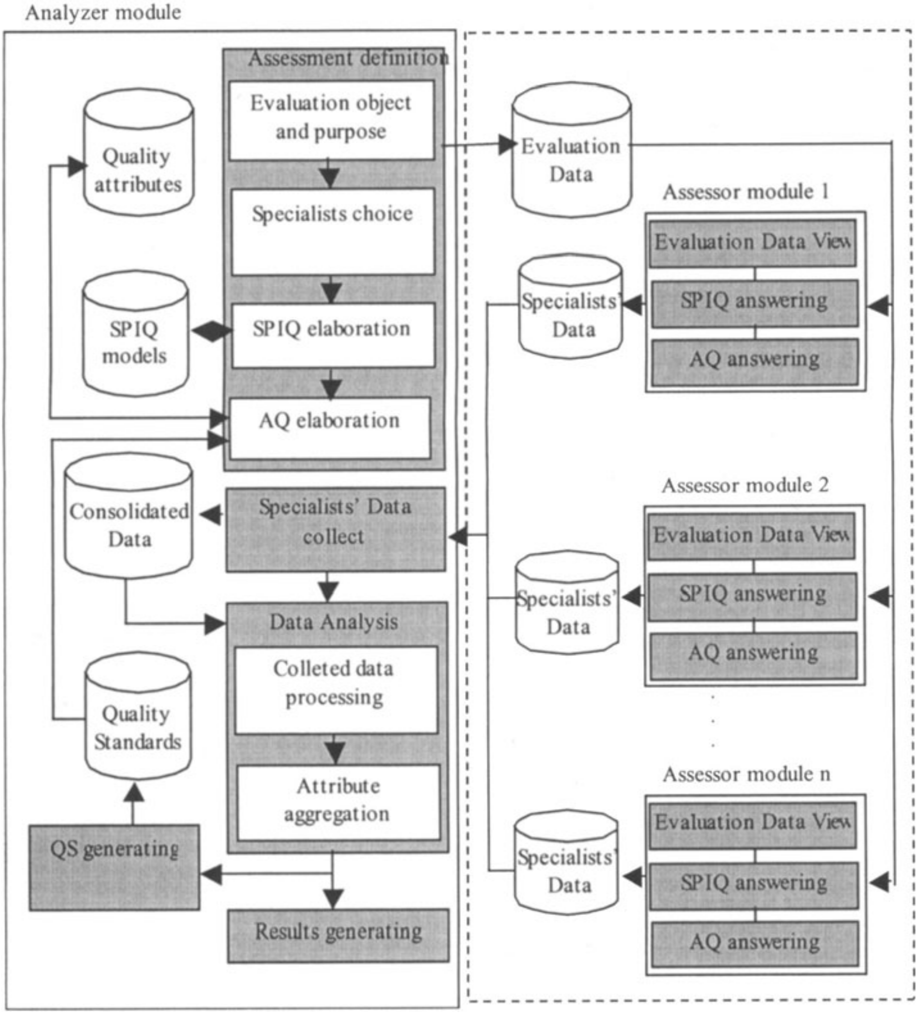


Figure 3. The *AdeQuaS* Fuzzy tool framework.

The characteristics and subcharacteristics contained on ISO/IEC 9126 [ISO 2001] are available in the tool, which can be used. Besides, it can be also used the quality standard characteristics of many application domains or other characteristics proposed by the responsible for assessment definition.

After the assessment definition, the modules *AdeQuaS*-Assessor, jointly the information derived from the data definition, are distributed among the specialists. At this moment, they can begin the evaluation. While the evaluators will be finishing their judgments, the data is sent and incorporated into the *AdeQuaS*-Analyzer.

At the end of evaluation collect, the data analysis could be executed through the data processing and the attribute aggregation. Those are the stages 4 and 5 of the FMSQE. After that, the evaluation results are finally concluded.

If the situation consists on importance degrees determination of a product or an application domain, the quality standard is generated at the end of the evaluation process. The quality standards will be used as a base to the next evaluations.

The QS generating provide more functionality, once they can be used to evaluate a software product with same or similar context. It is easy to identify because all generated QS are stored into categories that describe which subject they are about.

All the quality standards are categorized according to certain domain ontology, into AdeQuaS database. The categorization process is one of the activities in the beginning of the QS generation, which has the objective to characterize the application domain adequately and in detail. After the categorization, the following activity is to feed the quality attributes database. In this case, the AdeQuaS tool allows the partial or total reuse of each attribute stored in the database. At the end, it offers the options:

- The use of the attributes of software quality, created from ISO/IEC 9126 [ISO 2001], also called of standard hierarchical tree, which is presented as *default*;
- The reuse of attributes already stored, which has the same category of the product to be evaluated. For instance: let's consider that attributes of website quality and others of educational software are stored in database. If we intend to evaluate a educational Website, the AdeQuaS will suggest that the new AQ are created from the aforementioned ones, jointly with the standard hierarchical tree;
- The reuse of QS, which have some common characteristics, according to the categorization;
- The reuse of attributes stored, independently of their categorization.

4.2. The AdeQuaS-Analyzer Module

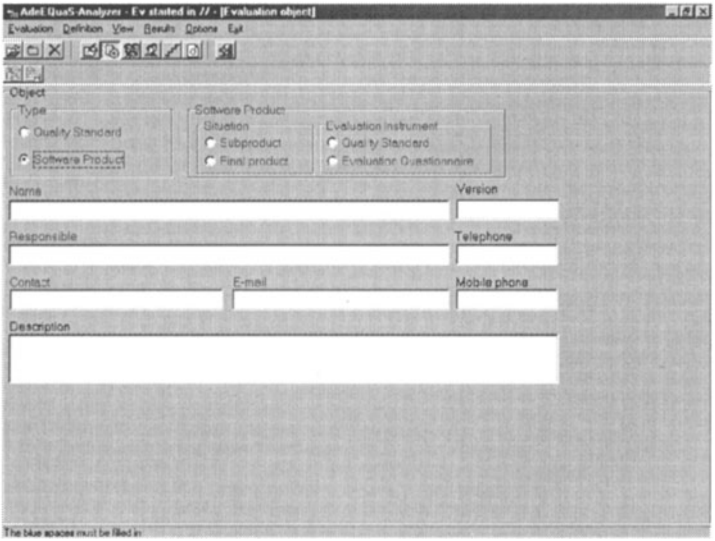
The module *AdeQuaS*-Analyzer can manage many evaluations at the same time. Each evaluation must be identified by a name and recorded in a chosen work directory.

The assessment definition includes: *(i)* evaluation data, *(ii)* object to be evaluated data, *(iii)* involved specialists record, *(iv)* SPIQ elaboration, *(v)* linguistic terms definition, and *(vi)* AQ elaboration.

The evaluation data includes assessment process initial date, responsible and general-purpose specification.

The object data, as shown in Figure 4, includes object name, version and responsible, as well as important features to the assessment evaluation, such as:

- *Object type*: could be specified whether the object is the quality standard establishment or the software product evaluation. This specification determines which situation the assessment is involved. In fact, it makes clear if the context is the *Situation 1* (as those described in section 3) of the FMSQE.
- *If it would be chosen the "Software Product" option, it must be filled the following options in*: *(i)* Software product situation: identifies whether it will be executed an assessment of an intermediate or a final product, and *(ii)* Instrument: determines whether the evaluation will be based on a predefined quality standard or an assessment questionnaire. It makes clear whether the context involves the *Situation 2* or *3* of the FMSQE.



Object

Type
○ Quality Standard
● Software Product

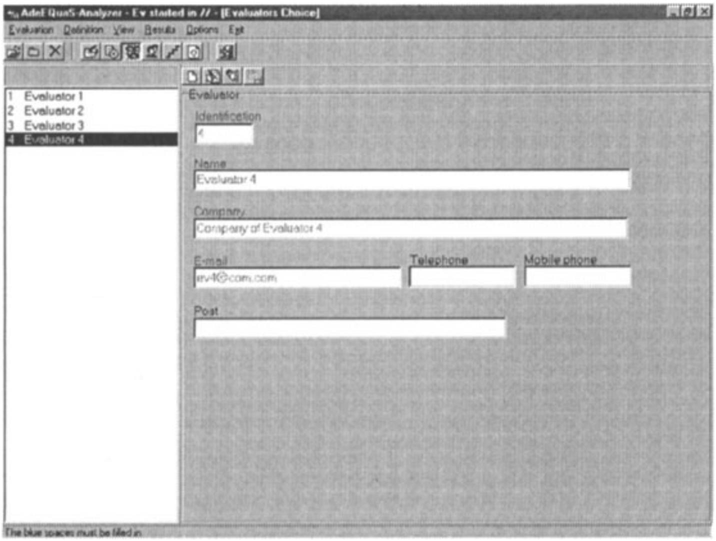
Software Product
Situation
○ Subproduct
○ Final product
Evaluation Instrument
○ Quality Standard
○ Evaluation Questionnaire

Name
Version
Responsible
Telephone
Contact
E-mail
Mobile phone
Description

The blue spaces must be filled in

Figure 4. Object definitions.

Then, the specialists must be chosen. The Figure 5 presents the specialists record window. Each specialist of the defined group to an evaluation will receive the *AdeQuaS*-Assessor in order to proceed with the judgment.


AdeEQuaS-Analyzer - Ev started in // - [Evaluators Choice]
Evaluation Definition View Results Options Exit

1 Evaluator 1
2 Evaluator 2
3 Evaluator 3
4 Evaluator 4

Evaluator

Identification: 4

Name: Evaluator 4

Company: Company of Evaluator 4

E-mail: ev4@com.com

Telephone:

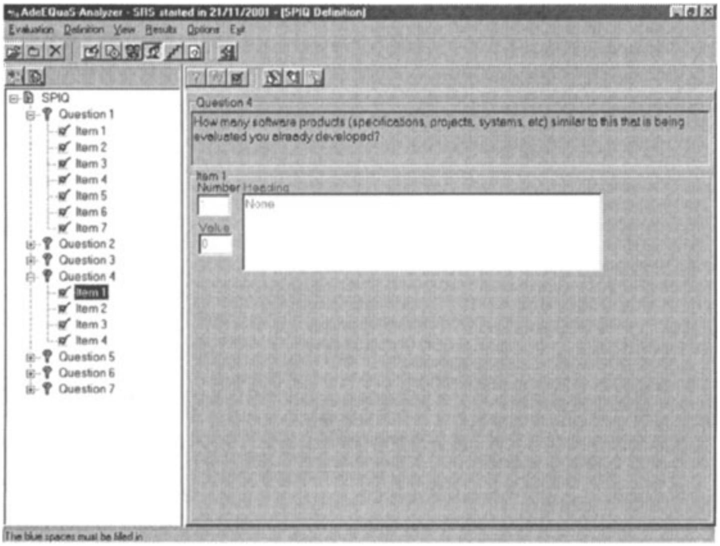
Mobile phone:

Post:

The blue spaces must be filled in

Figure 5. Specialists group choice.

The SPIQ elaboration is built according to a tree format with until two levels of questions, as instanced in Figure 6. Each item has an associated score that will be used to calculate the specialist's weight.


AdeEQuaS-Analyzer - SfIS started in 21/11/2001 - [SPIQ Definition]
Evaluation Definition View Results Options Exit

SPIQ
- Question 1
  - Item 1
  - Item 2
  - Item 3
  - Item 4
  - Item 5
  - Item 6
  - Item 7
- Question 2
- Question 3
- Question 4
  - Item 1
  - Item 2
  - Item 3
  - Item 4
- Question 5
- Question 6
- Question 7

Question 4

How many software products (specifications, projects, systems, etc) similar to this that is being evaluated you already developed?

Item 1

Number: 1

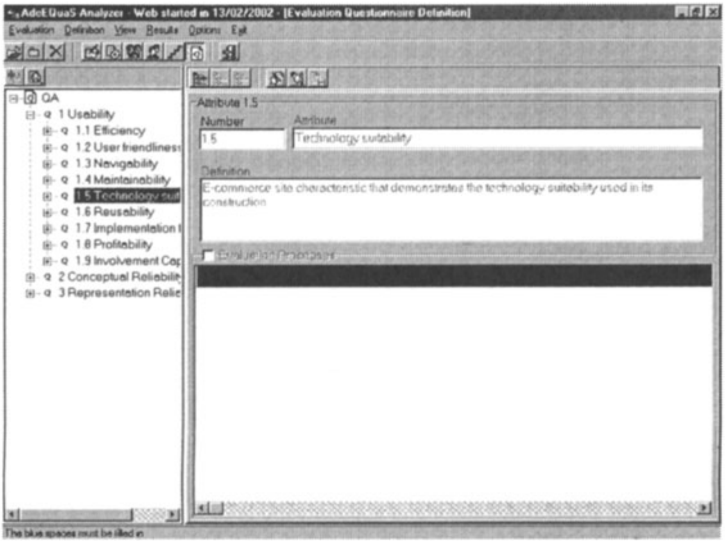
Heading: None

Value: 0

The blue spaces must be filled in

Figure 6. Elaboration of the Specialist Profile Identification Questionnaire.

The assessment questionnaire (AQ) consists of the quality requirements specification, related to the product in question, specified in a hierarchical way. The Figure 7 shows the record of e-commerce Website quality form, defined in [Albuquerque 2001], that obeys the hierarchical form of attributes.


AdeEQuaS Analyzer - Web started in 13/02/2002 - [Evaluation Questionnaire Definition]
Evaluation
Definition
View
Results
Options
Exit

QA
1 Usability
1.1 Efficiency
1.2 User friendliness
1.3 Navigability
1.4 Maintainability
1.5 Technology sui
1.6 Reusability
1.7 Implementation t
1.8 Profitability
1.9 Involvement Cap
2 Conceptual Reliabilit
3 Representation Relia

Attribute 1.5
Number: 1.5
Attribute: Technology suitability
Definition: E-commerce site characteristic that demonstrates the technology suitability used in its construction
□ Evaluation Proposal

The blue spaces must be filled in

Figure 7. Quality Requirements Specification.

By concluding the evaluation research, the results can be viewed, through the following options:

- *Evaluator profile*: views the specialist weight, according to the Stage 2 calculations of the FMSQE;
- *Attribute aggregation per level*: shows the aggregation results of a specific level of attributes tree. In this option, the Stage 5 calculations of the FMSQE are executed until the chosen level.

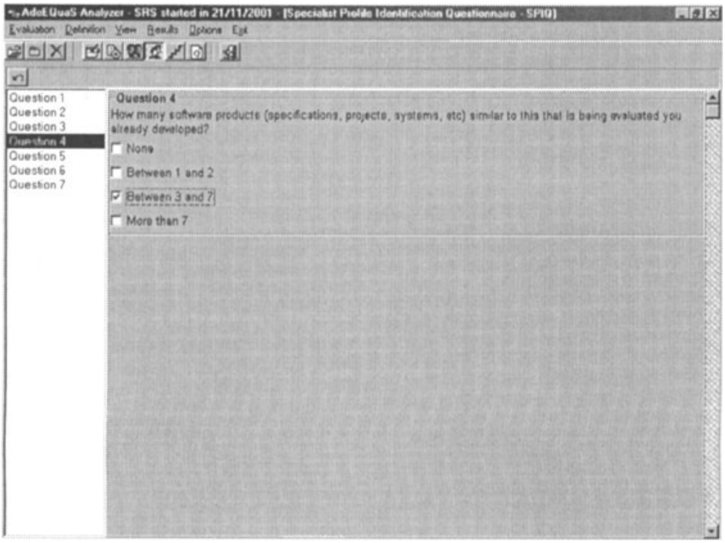
4.3. The AdeQuaS-Assessor Module

The module *AdeQuaS*-Assessor was developed to speed the assessment research. The research could be executed in many institutions, or in just one institution, with one or more evaluators.

The main objective is collecting the specialists' assessments, even if they would be geographically distributed, in order to the *AdeQuaS*-Analyzer could get the data appropriately.

The *AdeQuaS*-Assessor is basically a viewer. The assessment definition will be available to the specialists, in order to fill them in the context. The information is given as previously shown in Figure 4.

The evaluation is started with the SPIQ fulfilling, which will determine the importance of the specialists' opinion, based on their experience level. The SPIQ fulfilling is shown in Figure 8.


AdoEQuaS Analyzer - SRS started in 21/11/2001 - [Specialist Profile Identification Questionnaire - SPIQ]
Evaluation Definition View Results Options Exit
Question 1
Question 2
Question 3
Question 4
Question 5
Question 6
Question 7

Question 4

How many software products (specifications, projects, systems, etc) similar to this that is being evaluated you already developed?

□ None

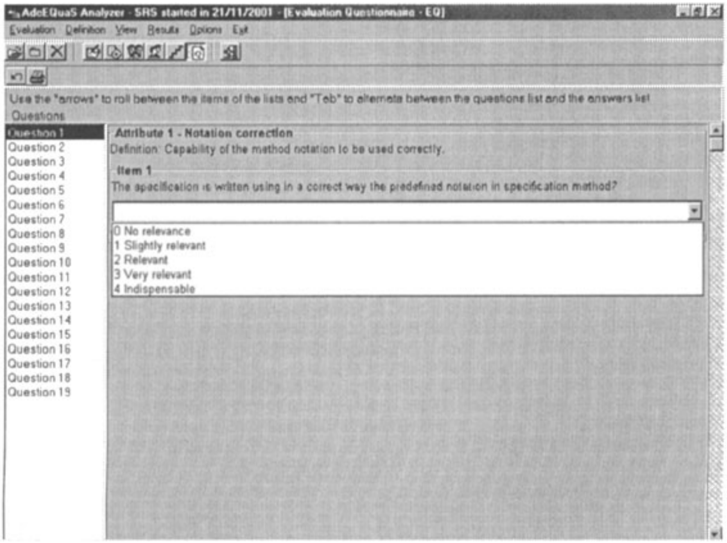
□ Between 1 and 2

☑ Between 3 and 7

□ More than 7

Figure 8. The SPIQ fulfilling.

Next, the attributes judgment is executed by the QA fulfilling, viewed in Figure 9.



Use the "arrows" to roll between the items of the lists and "Tab" to alternate between the questions list and the answers list

Attribute 1 - Notation correction

Definition: Capability of the method notation to be used correctly.

Item 1

The specification is writen using in a correct way the predefined notation in specification method?

0 No relevance
1 Slightly relevant
2 Relevant
3 Very relevant
4 Indispensable

Figure 9. The AQ fulfilling.

4.4. Case Study

Using the *AdeQuaS* Fuzzy tool, two evaluation processes was reproduced. The first was showed in [Belchior 1997] that refers to an evaluation of Software Requirements Specifications (SRS) . The second was developed in [Albuquerque 2001], which consists of establishment of a quality standard to e-commerce websites.

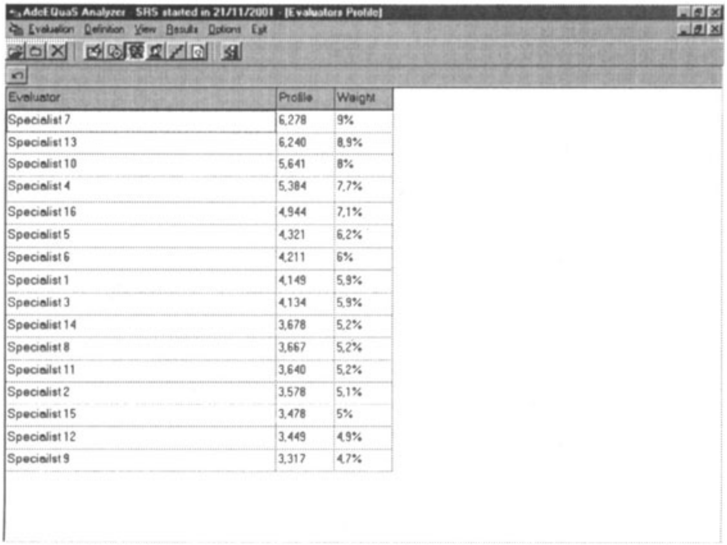
First Stage

They consist of the establishment of quality standards and the objective of the survey was to obtain from each selected specialist the degree of importance of each one of the relevant attributes, according to the application domain.

In the Software Requirements Specifications evaluation, 16 specialists, from 3 different institutions with a large experience in software development, participated in the process. On the other hand, the survey of the e-commerce websites evaluation was developed with 30 specialists.

Second Stage

The SPIQ has seven questions. The specialist's profiles and weights, which are obtained through the completion of the SPIQ, are presented in Figure 10 and Figure 11. These weighted values will be used in the aggregation stages, influencing in the final result of the evaluation. The values are showed in decreasing order of experience.



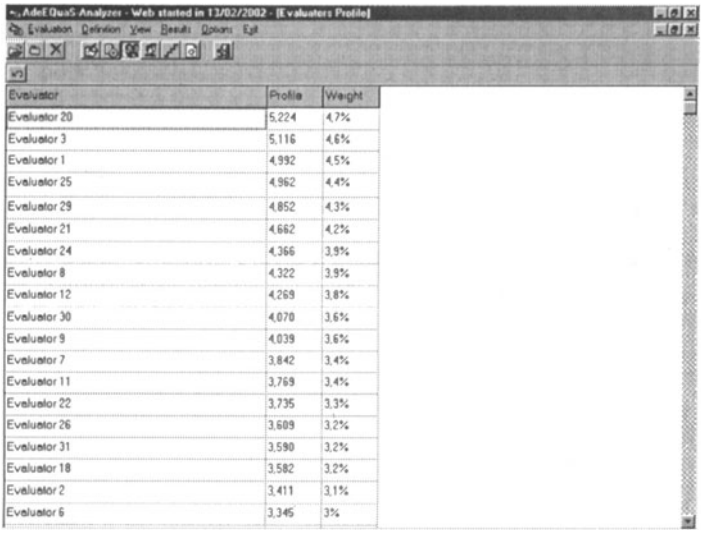
AdeEQuaS Analyzer - SRS started in 21/11/2001 - [Evaluators Profile]
Evaluation Definition View Results Options Exit

Evaluator	Profile	Weight
Specialist 7	6,278	9%
Specialist 13	6,240	8,9%
Specialist 10	5,641	8%
Specialist 4	5,384	7,7%
Specialist 16	4,944	7,1%
Specialist 5	4,321	6,2%
Specialist 6	4,211	6%
Specialist 1	4,149	5,9%
Specialist 3	4,134	5,9%
Specialist 14	3,678	5,2%
Specialist 8	3,667	5,2%
Speciailst 11	3,640	5,2%
Specialist 2	3,578	5,1%
Specialist 15	3,478	5%
Specialist 12	3,449	4,9%
Speciailst 9	3,317	4,7%

Figure 10. Specialists Profile Result on SRS evaluation.

Third Stage

The complete set of the quality attributes of the SRS is defined in [Belchior 1997] and of the e-commerce websites in [Albuquerque 2001]. The results are shown until two levels and they are classified as factors and subfactors. The subfactors are organized inside of the factors. The fuzzy numbers obtained after the execution of the stages of FMSQE aggregations are included.


AdeEQuaS-Analyzer - Web started in 13/02/2002 - [Evaluators Profile]
Evaluation Definition View Results Options Exit

Evaluator	Profile	Weight
Evaluator 20	5,224	4,7%
Evaluator 3	5,116	4,6%
Evaluator 1	4,992	4,5%
Evaluator 25	4,962	4,4%
Evaluator 29	4,852	4,3%
Evaluator 21	4,662	4,2%
Evaluator 24	4,366	3,9%
Evaluator 8	4,322	3,9%
Evaluator 12	4,269	3,8%
Evaluator 30	4,070	3,6%
Evaluator 9	4,039	3,6%
Evaluator 7	3,842	3,4%
Evaluator 11	3,769	3,4%
Evaluator 22	3,735	3,3%
Evaluator 26	3,609	3,2%
Evaluator 31	3,590	3,2%
Evaluator 18	3,582	3,2%
Evaluator 2	3,411	3,1%
Evaluator 6	3,345	3%

Figure 11. Specialists Profile Result on the e-commerce websites evaluation.

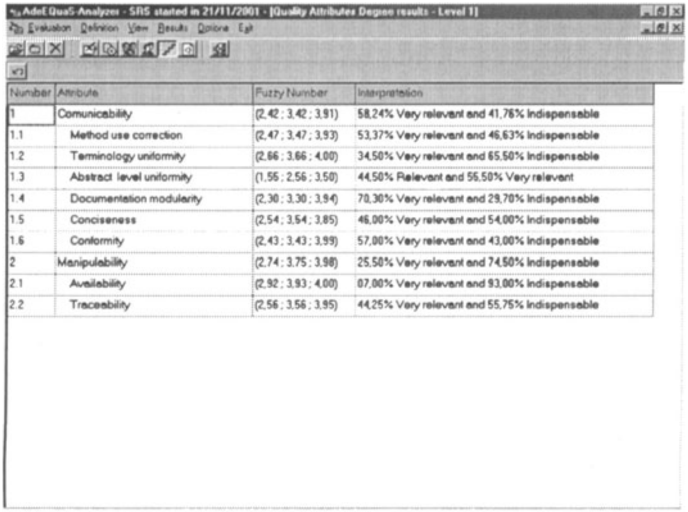
Fourth and Fifth Stages

The Table 4 presents a subset of quality attributes to Software Requirements Specifications (SRS) that is related to the objective “Representation Reliability”.

Software Quality Attributes to ERS	SRS to QS
Factor: Communicability	Ñ = (2.42, 3.42, 3.91)
Subfactor: Method use correction	Ñ= (2.47, 3.47, 3.93)
Subfactor: Terminology Uniformity	Ñ = (2.66, 3.66, 4.00)
Subfactor: Abstract Level Uniformity	Ñ = (1.55, 2.56, 3.50)
Subfactor: Documentation Modularity	Ñ = (2.30, 3.30, 3.94)
Subfactor: Conciseness	Ñ = (2.54, 3.54, 3.85)
Subfactor: Conformity	Ñ = (2.43, 3.43, 3.99)
Factor: Manipulability	Ñ = (2.74, 3.75, 3.98)
Subfactor: Availability	Ñ = (2.92, 3.93, 4.00)
Subfactor: Traceability	Ñ = (2.56, 3.56, 3.95)

Table 4. Quality Attribute Evaluation to SRS [Belchior 1997].

In [Belchior 1997], the results of the SRS evaluation were obtained with statistics methods. They were compared with the results presented by the tool (Figure 12) and we observed that *AdeQuaS* got results more precise.


AdeEQuaS-Analyzer - SRS started in 21/11/2001 - [Quality Attributes Degree results - Level 1]
Evaluation
Definition
View
Results
Options
Exit

Number	Attribute	Fuzzy Number	Interpretation
1	Comunicability	(2,42 ; 3,42 ; 3,91)	58,24% Very relevant and 41,76% Indispensable
1.1	Method use correction	(2,47 ; 3,47 ; 3,93)	53,37% Very relevant and 46,63% Indispensable
1.2	Terminology uniformity	(2,66 ; 3,66 ; 4,00)	34,50% Very relevant and 65,50% Indispensable
1.3	Abstract level uniformity	(1,55 ; 2,56 ; 3,50)	44,50% Relevant and 55,50% Very relevant
1.4	Documentation modularity	(2,30 ; 3,30 ; 3,94)	70,30% Very relevant and 29,70% Indispensable
1.5	Conciseness	(2,54 ; 3,54 ; 3,85)	46,00% Very relevant and 54,00% Indispensable
1.6	Conformity	(2,43 ; 3,43 ; 3,99)	57,00% Very relevant and 43,00% Indispensable
2	Manipulability	(2,74 ; 3,75 ; 3,98)	25,50% Very relevant and 74,50% Indispensable
2.1	Availability	(2,92 ; 3,93 ; 4,00)	07,00% Very relevant and 93,00% Indispensable
2.2	Traceability	(2,56 ; 3,56 ; 3,95)	44,25% Very relevant and 55,75% Indispensable

Figure 12. SRS Results View.

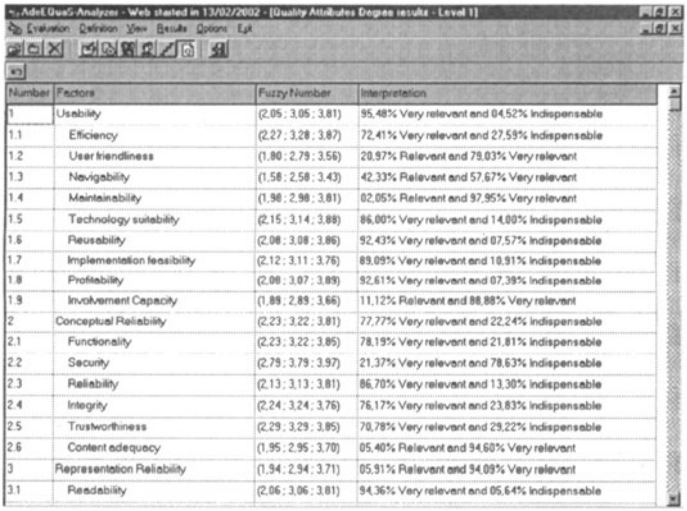
It is important to remark that the Availability subfactor was considered the most important. The result obtained indicates that, in Software Requirement Specifications, availability is the most relevant attribute. This tendency confirms the importance that many users could easily handle the specification in its updated version, through its development process. This subfactor obtained the defuzzification value of 3.93, which means 7% very relevant and 93% indispensable [Albuquerque, 2001]. In the same way, the Table 5 presents a set of quality attribute to e-commerce websites. The results presented by the tool appear in Figure 13.

E-commerce Websites Quality	Websites to QS
Factor: Usability	Ñ = (2.05, 3.05, 3.81)
Subfactor: Efficiency	Ñ= (2.27, 3.28, 3.87)
Subfactor: User friendliness	Ñ = (1.80, 2.79, 3.56)
Subfactor: Navigability	Ñ = (1.58, 2.58, 3.43)
Subfactor: Maintainability	Ñ = (2.98, 3.98, 3.81)
Subfactor: Technology suitability	Ñ = (2.15, 3.14, 3.88)
Subfactor: Reusability	Ñ = (2.08, 3.08, 3.86)
Subfactor: Implementation feasibility	Ñ = (2.12, 3.11, 3.76)
Subfactor: Profitability	Ñ = (2.08, 3.07, 3.89)
Subfactor: Involvement Capacity	Ñ = (1.89, 2.89, 3.66)
Factor: Conceptual Reliability	Ñ = (2.23, 3.22, 3.81)
Subfactor: Functionality	Ñ = (2.23, 3.22, 3.85)
Subfactor: Security	Ñ = (2.79, 3.79, 3.97)
Subfactor: Reliability	Ñ = (2.13. 3.13, 3.81)
Subfactor: Integrity	Ñ = (2.24, 3.24, 3.76)
Subfactor: Trustworthiness	Ñ = (2.29, 3.29, 3.85)
Subfactor: Content adequacy	Ñ = (1.95, 2.95, 3.70)
Factor: Representation Reliability	Ñ = (1.94, 2.94, 3.71)
Subfactor: Readability	Ñ = (2.06, 3.06, 3.81)
Subfactor: Standards conformance	Ñ = (2.99, 3.99, 3.75)
Subfactor: Easy of manipulation	Ñ = (1.72, 2.72, 3.53)

Table 5. Quality attribute evaluation to e-commerce websites quality [Albuquerque 2001].

This result portrays the web context, wherein electronic Commerce is inserted. It is important to remark that the Security subfactor was considered the most important. The result obtained indicates that, in e-commerce websites, security is fundamental, especially when it comes to electronic payments, which cannot be vulnerable to any kind of attack, and when it comes to the subject of site authentication itself.

The websites that don’t pay enough attention to those three items especially, which are taken as indispensable safety requirements, may not even be accessed by possible commercial transactions through the Internet, or may not be accessed later by potential users. This factor obtained the defuzzification value of 3.79, that is 21.37% very relevant and 78.63% indispensable [Albuquerque 2001].


AdeEQuaS-Analyzer - Web started in 13/02/2002 - [Quality Attributes Degree results - Level 1]
Evaluation
Definition
View
Results
Options
Exit

Number	Factors	Fuzzy Number	Interpretation
1	Usability	(2,05 ; 3,05 ; 3,81)	95,48% Very relevant and 04,52% Indispensable
1.1	Efficiency	(2,27 ; 3,28 ; 3,87)	72,41% Very relevant and 27,59% Indispensable
1.2	User friendliness	(1,80 ; 2,79 ; 3,56)	20,97% Relevant and 79,03% Very relevant
1.3	Navigability	(1,58 ; 2,58 ; 3,43)	42,33% Relevant and 57,67% Very relevant
1.4	Maintainability	(1,98 ; 2,98 ; 3,81)	02,05% Relevant and 97,95% Very relevant
1.5	Technology suitability	(2,15 ; 3,14 ; 3,88)	86,00% Very relevant and 14,00% Indispensable
1.6	Reusability	(2,08 ; 3,08 ; 3,86)	92,43% Very relevant and 07,57% Indispensable
1.7	Implementation feasibility	(2,12 ; 3,11 ; 3,76)	89,09% Very relevant and 10,91% Indispensable
1.8	Profitability	(2,08 ; 3,07 ; 3,89)	92,61% Very relevant and 07,39% Indispensable
1.9	Involvement Capacity	(1,89 ; 2,89 ; 3,66)	11,12% Relevant and 88,88% Very relevant
2	Conceptual Reliability	(2,23 ; 3,22 ; 3,81)	77,77% Very relevant and 22,24% Indispensable
2.1	Functionality	(2,23 ; 3,22 ; 3,85)	78,19% Very relevant and 21,81% Indispensable
2.2	Security	(2,79 ; 3,79 ; 3,97)	21,37% Very relevant and 78,63% Indispensable
2.3	Reliability	(2,13 ; 3,13 ; 3,81)	86,70% Very relevant and 13,30% Indispensable
2.4	Integrity	(2,24 ; 3,24 ; 3,76)	76,17% Very relevant and 23,83% Indispensable
2.5	Trustworthiness	(2,29 ; 3,29 ; 3,85)	70,78% Very relevant and 29,22% Indispensable
2.6	Content adequacy	(1,95 ; 2,95 ; 3,70)	05,40% Relevant and 94,60% Very relevant
3	Representation Reliability	(1,94 ; 2,94 ; 3,71)	05,91% Relevant and 94,09% Very relevant
3.1	Readability	(2,06 ; 3,06 ; 3,81)	94,36% Very relevant and 05,64% Indispensable

Figure 13. E-commerce websites quality results view.

This model has been satisfactory used to evaluate others application domains or software development stages, like:

- Software Project Management Quality Evaluation [Branco Jr. and Belchior 2001];
- Software Component Quality [Simão 2002].

5. CONCLUSION

The Fuzzy Model for Software Quality Evaluation has some relevant characteristics, which some of them are considered necessary by Bardossy *et al.* [1993]:

1. *Agreement preservation*: if all estimates are identical, the combined result will be the common estimate;
2. *Order independence*: the result does not depend on the order with which individual opinion or estimate are pooled;

3. *Joint influence of degree of concordance and specialist weight*: if a specialist have a small agreement degree the final weight attribute to his or her opinion will be smaller than the original weight correspondent to the specialist experience; and
4. *Fuzzy number preservation*: if all opinions are normal triangular fuzzy numbers, the aggregation will also be a normal triangular fuzzy number.

Some evaluation problems were minimized by the FMSQE, once it acts as an instrument to aggregate attributes, to get the evaluators' consensus and to get the degree that represents quantitatively the software quality level. Although this model is flexible to be used in many situations, its use requires a reasonable effort.

The *AdeQuaS* Fuzzy tool, an evaluation process automation fuzzy tool based on the FMSQE, makes transparent to the participants most arduous tasks in the model. Also it brings a greater functionality on FMSQE use, allowing the use of stored quality characteristics and quality standards.

Besides of a quicker evaluation application and result generation, the tool makes possible to obtain more accurate and reliable results. The presented benefits increased the practicability of the process execution, promoting greater trust in the quality improvement.

The e-commerce websites evaluation, which was executed as an application of the *AdeQuaS* Fuzzy tool, allows analyzing the importance of its relevant characteristics, confirming that the factor Security is one of the most important characteristics with the best score of the evaluation.

REFERENCES

Albuquerque, A. B. (2001), "Electronic Commerce Websites Quality". MSc Thesis, Department of Computer Science, University of Fortaleza, Fortaleze, CE. (in Portuguese).

Araújo, K. (2000), "Fuzzy Logic: history, concepts and fuzzy model applications", Developers Magazine, April, 28-33 (in Portuguese).

Baldwin, J. F. (1979), "A new approach to approximate reasoning using a fuzzy logic", Fuzzy Sets and Systems 2, 309-325.

Bardossy, A., Duckstein. L. and Bogardi, I. (1993), "Combination of fuzzy number representing expert opinions",. Fuzzy Sets and Systems 57, 173-181.

Belchior, A. D. (1997), "A Fuzzy Model for Software Quality Evaluation", DSc Thesis, Department of Systems Engineering and Computer Science, Federal University of Rio de Janeiro, RJ (in Portuguese).

Boehm, B. and Hoh, I. (1996), "Identifying Quality Requirements Conflicts",. IEEE Software, 25-35.

Boloix, G. et.al. (1995), "A software system evaluation framework", IEEE Software, 17-26.

Branco Jr., E. C., Belchior, A. C. (2001), "Management processes of software projects: a qualitative approach", In VIII Software Quality Workshop, Rio de Janeiro, RJ.

Campos, F. et al. (1998), "Farming software quality: a user's view", IX International Conference of Software Technology, Curitiba, PR (in Portuguese).

Chen, C. T., Hsy, H. M. (1993), A study of fuzzy TOPSIS model, Proc. of the Chinese Institute of Industrial Engineers National Conference, in (Hsu, 1996).

Clunie, C. E. (1997) Quality Evaluation of Object-Oriented Specifications. DSc. Thesis, Department of Systems Engineering and Computer Science, Federal University of Rio de Janeiro, RJ (in Portuguese).

Dubois, D. and Prade, H. (1980), Fuzzy Sets and Systems: Theory and Applications, Academic Press, NY.

Dubois, D. and Prade, H. (1991), "Fuzzy sets in approximate reasoning. Part 1: Inference with possibility distributions", Fuzzy Sets and Systems 40, IFSA, Special Memorial Volume: 25 years of fuzzy sets, North-Holland, Amsterdam, 143-202.

Dyer, M. (1992), The cleanroom approach to Quality Software Development, John Wiley & Sons, Inc. NY.

Fenton, N. E. and Pfleeger, S. L. (1997), Software Metrics: a rigorous & practical approach, Second Edition, PWS Publishing Company, Boston, MA.

Idri, A. et.al. (2000), "COCOMO – Cost model using fuzzy logic", In 7th International Conference on Fuzzy Theory & Technology. Atlantic City, NJ.

Gray, A. (1997), "Applications of fuzzy logic to software metric models for development effort estimation", In Proceedings of the 1997 Annual Meeting of the North American Fuzzy Information Processing Society. IEEE Computer Society Press, Syracuse, NY, pp. 394-399.

Hapke, M. et.al. (1994), "Fuzzy project scheduling system for software development", Fuzzy Sets and Systems 67, 101-117.

Hsu, H. M. and Chen, C. T (1996), "Aggregation of fuzzy opinions under group decision making", Fuzzy Sets and Systems 79, 279-285.

Ibrahim, A. and Ayyub, B. M. (1992), "Multi-criteria ranking of components according to their priority for inspection", Fuzzy Sets and Systems 48, 1-14.

ISO (2001), ISO/IEC 9126-1, "Software engineering – Product quality – Part 1: quality model".

ISO (1998), ISO/IEC 14598-1, "Information technology – software product evaluation – Part 1: general overview".

Ishikawa, A. et.al. (1993), "The max-min Delphi method and fuzzy Delphi method via fuzzy integretion", Fuzzy Sets and Systems 55, 241-253.

Kacprzyk, J. et.al. (1992), "Group decision making and consensus under fuzzy preference and fuzzy majority", Fuzzy Sets and Systems 49, 21-31.

Kaufmann, A. and Gupta, M. M. (1991), Introduction to Fuzzy Arithmetic: theory and applications. Van Nostrand Reinhold, NY.

Kitchenham, B. et.al. (1996), "Software Quality: the elusive target", IEEE Software, 12-21.

Lasek, M. (1992), "Hierarchical structures of fuzzy ratings in the analysis of strategic goal of enterprises", Fuzzy Sets and Systems 50, 127-134.

Lee, H. M. (1996a), "Applying fuzzy set theory to evaluate the rate of aggregative risk in software development", Fuzzy Sets and Systems 79, 323-336.

Lee, H. M. (1996b), "Group decision making using fuzzy theory for evaluating the rate of aggregative risk in software development", Fuzzy Sets and Systems 80, 261-271.

Lima Jr. O. S. et.al. (2001), "Maintenance project assessments using fuzzy function point analysis", Seventh Workshop on Empirical Studies of Software Maintenance, IEEE Computer Society, Florence, Italy, 114-121.

Liou, T. S., Jiun, M. and WG, J. (1992), "Fuzzy weighted average: an improved algorithm", Fuzzy Sets and Systems 49, 307-315.

Möller, K. H. (1993), Software metrics: a practitioner's guide to improved product development, Chapman & Hall Computing, London, England.

Palermo, S. and Rocha, A. R. C (1989), "An experience on evaluating software quality for high energy physics", Computer Physics Communications.

Pedrycz, W. and Peters, J. F. (1998) "Software Quality Measurement: concepts and fuzzy neural relational model", http://neuron.et.ntust.edu.tw/homework/89/FL/89homework/M8709022/3.pdf.

Pfleeger, S. L. (1998) Software Engineering: theory and practice, Prentice Hall, NJ.

Pressman, R. S. (2000), Software engineering: a practitioner's approach. Fifth Edition. McGraw Hill, NY.

Ribeiro, R. A. (1996), "Fuzzy multiple attribute decision making: a review and new preference elicitation techniques", Fuzzy Sets and Systems 78, 155-181.

Rocha, A. R. C. (1983), "A model for specification quality evaluation", D.Sc. Thesis, Department of Systems Engineering and Computer Science, Pontifical Catholic University (PUC), Rio de Janeiro, RJ (in Portuguese).

Rocha, A. R. C. et al. (2001), Software Quality: theory and practice, Prentice Hall, São Paulo, Brazil. (in Portuguese).

Römer, C. and Kandel, A. (1995), "Statistical tests for fuzzy data", Fuzzy Sets and Systems 72, 1-26.

Ruoning, X. and Xiaoyan, Z. (1992), "Extensions of the analytic hierarchy process in fuzzy environment", Fuzzy Sets and Systems 52, 251-257.

Ryder, J. (1998) "Fuzzy modeling of software effort prediction.", IEEE Information Technology Conference, Syracuse, NY, pp 53-56.

Schneidewind, N. F. (1992), "Methodology for validating software metrics", IEEE Transaction Software Engineering, vol. 18, n° 5, May, 1992, in (Fenton, 1994).

Simão, R. P. S. (2002), "A quality standard to software components", In I Brazilian Symposium of Software Quality , Gramado, RS, 249-260.

Simonelli, M. R. (1996), "On fuzzy interactive knowledge", Fuzzy Sets and Systems 80, 159-165.

Sousa, C. P. (1995) "Fuzzy Logic Introduction". http://dee.ufe.br/~pimentel/ica/ica.html. (in Portuguese).

Zadeh, L. A. (1988), "Fuzzy Logic", IEEE Transaction Computer., vol. 21, 83-93.

Zimmermann, H. J. (1996), Fuzzy Set Theory and Its Applications, Third Edition, Kluwer Academic Publishers, Boston, MA.

Zwick, R., Edward Carlstein and David V. Budescu (1987), "Measures of similarity among fuzzy concepts: A comparative analysis," International Journal of Approximate Reasoning 1. 221-242.

Software Quality Prediction Using Bayesian Networks

Martin Neil[1], Paul Krause[2] and Norman Fenton[1]

[1]*Queen Mary*
University of London and Agena Ltd., UK
martin@dcs.qmul.ac.uk

[2]*Department of Computing*
University of Surrey and Philips Research Laboratories, UK

ABSTRACT

Although a number of approaches have been taken to quality prediction for software, none have achieved widespread applicability. Our aim here is to produce a single model to combine the diverse forms of, often causal, evidence available in software development in a more natural and efficient way than done previously. We use Bayesian Belief Networks as the appropriate formalism for representing this evidence. We can use the subjective judgements of experienced project managers to build the probability model and use this model to produce forecasts about the software quality throughout the development life cycle. Moreover, the causal or influence structure of the model more naturally mirrors the real world sequence of events and relations than can be achieved with other formalisms. The paper focuses on the particular model that has been developed for Philips Consumer Electronics, using expert knowledge from Philips Research Labs. The model is used especially to predict defect rates at various testing and operational phases. To make the model usable by software quality managers we have developed a tool (AID) and have used it to validate the model on 28 diverse projects from within Philips. In each of these projects, extensive historical records were available. The results of the validation are encouraging. In most cases the model provides accurate predictions of defect rates even on projects whose size was outside the original scope of the model.

1. INTRODUCTION

Important decisions need to be made during the course of developing software products. Perhaps the most important of these is the decision when to release the software product. The consequences of making an ill-judged decision can be potentially critical for the reputation of a product or its supplier. Yet, such decisions are often made informally, rather than on the basis of more objective and accountable criteria.

Software project and quality managers must juggle a combination of uncertain factors, such as use of tools, skill and experience level of personnel, development methods and testing strategies to achieve the delivery of a quality product to budget and on time. Each of these uncertain factors influences the introduction, detection and correction of defects at all stages in the development life cycle from initial requirements to product delivery.

In order to achieve software quality during development special emphasis needs to be applied to the following three activities in particular:

- Defect prevention;
- Defect detection;
- Defect correction.

The decision challenge during software development is to apply finite resources to all of these activities, and based on the division of resources applied, predict the likely quality that will be achieved when the product is delivered. To date the majority of software projects have tended to rely upon the judgement of the project or quality manager. Unfortunately, where mathematical or statistical procedures have been applied their contribution has been marginal at best [Fenton and Neil, 1999]. We will briefly outline the problems with current approaches in Section 2.

Our aim here is to extend the work introduced in [Fenton and Neil, 1999] and produce a single model to combine the diverse forms of, often causal, evidence available in software development in a more natural and efficient way than done previously. We use graphical probability models (also known as Bayesian Belief Networks) as the appropriate formalism for representing this evidence. We can use the subjective judgements of experienced project managers to build the probability model and use this model to produce forecasts about the software quality throughout the development life cycle. Moreover, the causal or influence structure of the model more naturally mirrors the real world sequence of events and relations than can be achieved with other formalisms.

After outlining the problems with current approaches to defect prediction, we will provide an introduction to probabilistic modelling. We will then describe the probabilistic model for defect prediction that has been built for use in Philips software development organisations, and provide results from initial validation studies.

2. THE PROBLEMS WITH SOFTWARE DEFECT PREDICTION

In this paper we examine the general issues relating to software defect prediction. However, it is worth phrasing the problem in general terms to emphasise that the longer-term goal is to apply Bayesian Networks (BNs) to other quality characteristics, like reliability and safety [Neil et al 1996, Fenton and Neil 2000].

There are two different viewpoints of software quality as defined by Fenton and Pfleeger [Fenton and Pfleegar 1997]. The first, the external product view, looks at the characteristics and sub-characteristics that make up the user's perception of quality in the final product – this is often called quality-in-use. Quality-in-use is determined by measuring external properties of the software, and hence can only be measured once the software product is complete. For instance quality here might be defined as freedom from defects or the probability of executing the product, failure free, for a defined period.

The second viewpoint, the internal product view, involves criteria that can be used to control the quality of the software as it is being produced and that can form early predictors of external product quality. Good development processes and well-qualified staff working on a defined specification are just some of the pre-requisites for producing a defect free product. If we can ensure that the process conditions are right, and can check intermediate products to ensure this is so, then we can perhaps produce high quality products in a repeatable fashion.

Unfortunately the relationship between the quality of the development processes applied and the resulting quality of the end products is not deterministic. Software development is a profoundly intellectual and creative design activity with vast scope for error and for differences in interpretation and understanding of requirements. The application of even seemingly straightforward rules and procedures can result in highly variable practices by individual software developers. Under these circumstances the relationships between internal and external quality are uncertain. Typically informal assessments of critical factors will be used during software development to assess whether the end product is likely to meet requirements:

- Complexity measures: A complex product may indicate problems in the understanding of the actual problem being solved. It may also show that the product is too complex to be easily understood, de-bugged and maintained. Assessing the complexity of software programs and designs has remained central to much of software measurement since the topic's inception in the early 1970s. Useful examples of empirical validation of complexity measures can be found in [Cartwright and Shepperd 1997, Basili et al 1996, Koshgoftaar and Munson 1990].

- Process maturity: Development processes that are chaotic and rely on the heroic efforts of individuals can be said to lack maturity and will be less likely to produce quality products, repeatedly.
- Test results: Testing products against the original requirements can give some indication of whether they are defective or not. However the results of the testing are likely only to be as trustworthy as the quality of the testing done.

The above types of evidence are often collected in a piecemeal fashion and used to inform the project or quality manager about the quality of the final product. However there is often no formal attempt, in practice, to combine these evidences together into a single quality model.

A holy grail of software quality control could be the identification of one simple internal product measurement that provides an advanced warning of whether or not the goals for the external product characteristics will be achieved. Unfortunately, in software engineering the causal relationships between internal and external quality characteristics are rarely straightforward. We will illustrate this with one simple example. More detailed analyses of naïve regression models for software engineering can be found in [Fenton and Neil 1999], and [Fenton and Ohlsson 2000].

Suppose we have a product that has been developed using a set of software modules. A certain number of defects will have been found in each of the software modules during testing. Perhaps we might assume that those modules that have the highest number of defects during testing would have the highest risk of causing a failure once the product was in operation? That is, we might expect to see a relationship similar to that shown in Figure 2.1.

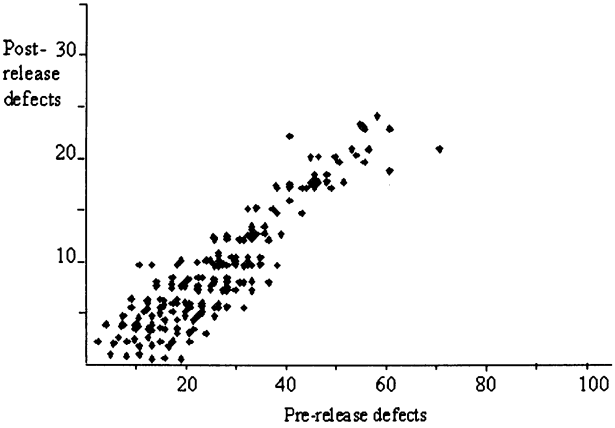


Figure 2.1. A hypothetical plot of pre-release against post-release defects for a range of modules. Each dot represents a module.

What actually happens? It is hard to be categorical. However, two published studies indicate quite the opposite effect – those modules that were most problematic pre-release had the least number of faults associated with them post-release. Indeed, many of the modules with a high number of defects pre-release showed zero defects post-release. This effect was first demonstrated by [Adams 1984], and replicated by [Fenton and Ohlsson 2000]. Figure 2.2 is an example of the sort of results they both obtained.

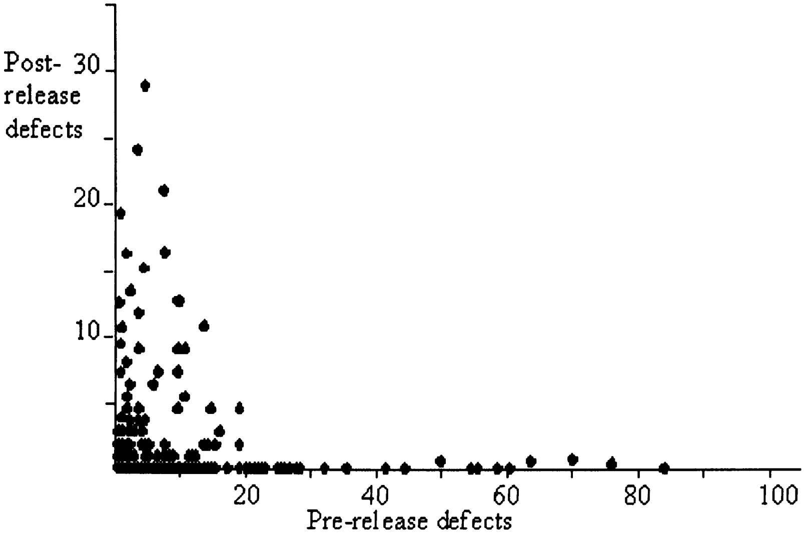


Figure 2.2. Actual plot of pre-release against post-release defects for a range of

modules.

So, how can this be? The simple answer is that faults found pre-release gives absolutely no indication of the level of residual faults unless the prediction is moderated by some measure of test effectiveness. In both of the studies referenced, those modules with the highest number of defects pre-release had had all their defects “tested out”. In contrast, many of the modules that had few defects recorded against them pre-release clearly turned out to have been poorly tested – they were significant sources of problems in the final implemented system.

Typically, the search is for a simple relationship between some predictor and the number of defects delivered. Size or complexity measures are often used as such predictors. The result is a “naïve” model that could be represented by the graph of Figure 2.3.

The difficulty is that whilst such a model can be used to *explain* a data set obtained in a specific context, none has so far been subject to the form of controlled statistical experimentation needed to establish a *causal* relationship. Indeed, the analysis of Fenton and Neil suggests that these models fail to include all the causal or explanatory variables needed in order to make the models generalisable. Further strong empirical support for these arguments is demonstrated in [Fenton and Ohlsson, 2000].

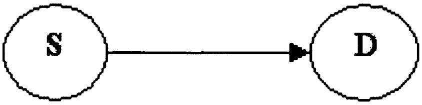


Figure 2.3. Graphical representation of a naïve regression model between some predictor S (typically a size measure), and the number of software defects D.

The model of Figure 2.3 can simulate the model of Figure 2.4 under certain circumstances. However, the latter has greater explanatory power, and can lead to quite a different interpretation of a set of data. One could take “Smoking” and “Higher Grades” at high school as an analogy. Just looking at the covariance between the two variables, we might see a correlation between smoking and achieving higher grades. However, if "Age" is then included in the model, we could have a very different interpretation of the same data. As a student's age increases, so does the likelihood of their smoking. As they mature, their grades also typically improve. The covariance is explained. However, for any fixed age group, smokers may achieve lower grades than non-smokers.

We believe that the relationships between product and process attributes and numbers of defects are too complex to admit straightforward curve fitting models. In predicting defects discovered in a particular project, we would certainly want to add additional variables to the model of Figure 2.4. For example, the number of defects discovered will depend on the effectiveness with which the software is tested. It may also depend on the level of detail of the specifications from which the test cases are derived, the care with which requirements have been managed during product development, and so on.

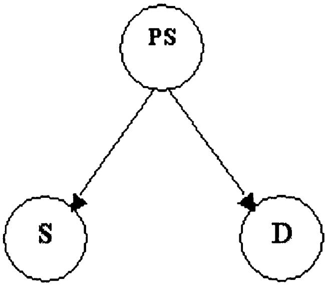


Figure 2.4. The influence of S on D is now mediated through a common cause PS. This model can behave in the same way as that of Figure 2.3, but only in certain specific circumstances.

We believe that graphical probabilistic models are the best candidate for situations with such a rich causal structure. Our primary reason for saying this is that we believe the influences on, for example, the presence of residual defects are too complex and varied to allow the development of effective and generalisable regressions using the sparse data that is available in the software engineering domain. Instead, we propose an alternative approach in which expert judgement can be used to help develop an initial model. This model can then be refined and revised while in use, to improve the accuracy of its predictions.

3. INTRODUCTION TO BAYESIAN NETWORKS

3.1. Conditional probability

The foundation for Bayesian Networks (BNs) is probability theory. Probabilities conform to three basic axioms:

- p(A), the probability of an event (outcome/consequence…), A, is a number between 0 and 1;
- p(A) = 0 means A is impossible, p(A)=1 means A is certain;
- p(A or B) = p(A) + p(B) provided A and B are disjoint.

However, merely to refer to the probability p(H) of an event or hypothesis is an oversimplification. In general, probabilities are context sensitive. For example, the probability of suffering from certain forms of cancer is higher in Europe than it is in Asia. Strictly, the probability of any event or hypothesis is conditional on the available evidence or current context. This can be made explicit by the notation p(H | E), which is read as "the probability of H given the evidence E". In the coin example, H would be a "heads" event and E an explicit reference to the evidence that the coin is a fair one. If there was evidence E' that the coin was double sided heads, then we would have p(H | E') = 1.0.

As soon as we start thinking in terms of conditional probabilities, we begin to need to think about the structure of problems as well as the assignment of numbers. To say that the probability of an hypothesis is conditional on one or more items is to identify the information relevant to the problem at hand. To say that the identification of an item of evidence influences the probability of an hypothesis being valid is to place a directionality on the links between evidences and hypotheses.

Often a direction corresponding to causal influence can be the most meaningful. For example, in medical diagnosis one can in a certain sense say that measles "causes" red spots (there might be other causes). So, as well as assigning a value to the conditional p('red spots' | measles), one might also wish to provide an explicit graphical representation of the problem. In this case it is very simple (Figure 3.1).

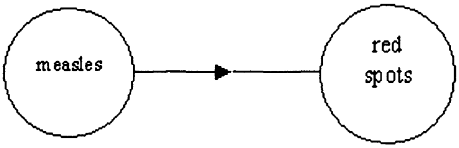


Figure 3.1. A very simple probabilistic network.

Note that to say that p('red spots' | measles) = *p* means that we can assign probability *p* to 'red spots' if measles is observed and only measles is observed. If any further evidence E is observed, then we will be required to determine p('red spots' | measles, E). The comma inside the parentheses denotes conjunction.

Building up a graphical representation can be a great aid in framing a problem. A significant recent advance in probability theory has been the demonstration of a formal equivalence between the structure of a graphical model and the dependencies that are expressed by a numerical probability distribution. In numerical terms, we say that event A is independent of event B if observation of B makes no difference to the probability that A will occur: p(A | B) = p(A). In graphical terms we indicate that A is independent of B by the absence of any direct arrow between the nodes representing A and B in a graphical model.

So far, we have concentrated on the static aspects of assessing probabilities and indicating influences. However, probability is a dynamic theory; it provides a mechanism for coherently revising the probabilities of events as evidence becomes available. Conditional probability and Bayes' Theorem play a central role in this. We will use a simple example to illustrate Bayesian updating, and then introduce Bayes' Theorem in the next section.

Suppose we are interested in the number of defects that are detected and fixed in a certain testing phase. If the software under test had been developed to high standards, perhaps undergoing formal reviews before release to the test phase, then the high quality of the software would in a sense "cause" a low number of defects to be detected in the test phase. However, if the testing were ineffective and superficial, then this would provide an alternative cause for a low number of defects being detected during the test phase. (This was precisely the common empirical scenario identified in [Fenton and Ohlsson, 2000]).

This situation can be represented by the simple graphical model of Figure 3.2. Here the nodes in the graph could represent simple binary variables with states "low" and "high", perhaps. However, in general a node may have many alternative states or even represent a continuous variable. We will stay with the binary states for ease of discussion.

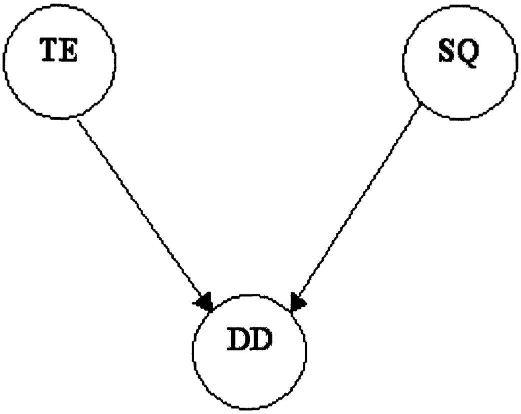


Figure 3.2. Some subtle interactions between variables captured in a simple graphical model. Node TE represents "Test Effectiveness", SQ represents "Software Quality" and DD represents "Defects Detected.

It can be helpful to think of Figure 3.2 as a fragment of a much larger model. In particular, the node SQ ("Software Quality") could be a synthesis of, for example: review effectiveness; developer's skill level; quality of input specifications; and, resource availability. With appropriate probability assignments to this model, a variety of reasoning styles can be modelled. A straightforward reasoning from cause to effect is possible. If TE (test effectiveness) is "low", then the model will predict that DD (defects discovered and fixed) will also be low. If earlier evidence indicates SQ (software quality) is "high", then again DD will be "low".

However, an important feature is that although conditional probabilities may have been assessed in terms of effect given cause, Bayes' rule enables inference to be performed in the "reverse" direction – to provide the probabilities of potential causes given the observation of some effect. In this case, if DD is observed to be "low" the model will tell us that low test effectiveness or high software quality are possible explanations (perhaps with an indication as to which one is the most likely explanation). The concept of "explaining away" will also be modelled. For example, if we also have independent evidence that the software quality was indeed high, then this will provide sufficient explanation of the observed value for DD and the probability that test effectiveness was low will be reduced.

This situation can be more formally summarised as follows. If we have no knowledge of the state DD then nodes TE and SQ are marginally independent – knowledge of the state of one will not influence the probability of the other being in any of its possible states. However, nodes TE and SQ are conditionally dependent given DD – once the state of DD is known there is an influence (via DD) between TE and SQ as described above.

We will see in the next section that models of complex situations can be built up by composing together relatively simple local sub-models of the above kind (See also [Neil *et al*, 2000]). This is enormously valuable. Without being able to structure a problem in this way it can be virtually impossible to assess probability distributions over large numbers of variables. In addition, the computational problem of updating such a probability distribution given new evidence would be intractable.

3.2. Bayes' theorem and Conditional Dependence

As indicated in the previous section, probability is a dynamic theory; it provides a mechanism for coherently revising the probabilities of events as evidence becomes available. Bayes' theorem is a fundamental component of the dynamic aspects.

As mentioned earlier, we write p(A | B) to represent the probability of some event (an hypothesis) conditional on the occurrence of some event B (evidence). If we are counting sample events from some universe Ω, then we are interested in the fraction of events B for which A is also true. In effect we are focusing attention from the universe Ω to a restricted subset in which B holds. From this it should be clear that (with the comma denoting conjunction of events):

$$p(A \mid B) = \frac{p(A, B)}{p(B)}$$

This is the simplest form of Bayes' rule. However, it is more usually rewritten in a form that tells us how to obtain a posterior probability in a hypothesis A after observation of some evidence B, given the *prior* probability in A and the likelihood of observing B were A to be the case:

$$p(A \mid B) = \frac{p(B \mid A)p(A)}{p(B)}$$

This theorem is of immense practical importance. It means that we can reason both in a forward direction from causes to effects, and in a reverse direction (via Bayes' rule) from effects to possible causes. That is, both deductive and abductive modes of reasoning are possible.

However, two significant problems need to be addressed. Although in principle we can use generalisations of Bayes' rule to update probability distributions over sets of variables, in practice:

1. Eliciting probability distributions over sets of variables is a major problem. For example, suppose we had a problem describable by seven variables each with two possible states. Then we will need to elicit (2^7-1) distinct values in order to be able to define the probability distribution completely. As can be seen, the problem of knowledge elicitation is intractable in the general case.
2. The computations required to update a probability distribution over a set of variables are similarly intractable in the general case.

Up until the late 1980's, these two problems were major obstacles to the rigorous use of probabilistic methods in computer based reasoning models. However, work initiated by Lauritzen and Spiegelhalter [1988] and Pearl [1988] provided a resolution to these problems for a wide class of problems. This work related the independence conditions described in graphical models to factorisations of the joint distributions over sets of variables. We have already seen some simple examples of such models in the previous section. In probabilistic terms, two variables X and Y are independent if p(X,Y) =

p(X)p(Y) – the probability distribution over the two variables factorises into two independent distributions. This is expressed in a graphic by the *absence* of a direct arrow expressing influence between the two variables.

We could introduce a third variable Z, say, and state that “X is conditionally independent of Y given Z”. This is expressed graphically in Figure 3.3. An expression of this in terms of probability distributions is:

$$p(X,Y \mid Z) = p(X \mid Z)p(Y \mid Z)$$

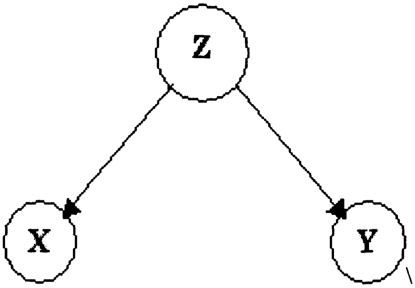


Figure 3.3. X is conditionally independent of Y given Z.

A significant feature of the graphical structure of Figure 3.3 is that we can now decompose the joint probability distribution for the variables X, Y and Z into the product of terms involving at most two variables:

$$p(X,Y,Z) = p(X \mid Z)p(Y \mid Z)p(Z)$$

In a similar way, we can decompose the joint probability distribution for the variables associated with the nodes DD, TE and SQ of Figure 4.2 as

$$p(DD, TE, SQ) = p(DD \mid TE,SQ)p(TE)p(SQ)$$

This gives us a series of example cases where a graph has admitted a simple factorisation of the corresponding joint probability distribution. If the graph is directed (the arrows all have an associated direction) and there are no cycles in the graph, then this property is a general one. Such graphs are called Directed Acyclic Graphs (DAGs). Using a slightly imprecise notation for simplicity, we have [Lauritzen and Spiegelhalter, 1988]:

3.2.1. Proposition

Let U = $\{X_1, X_2, \ldots, X_n\}$ have an associated DAG G. Then the joint probability distribution p(U) admits a direct factorisation:

$$p(U) = \prod_{i=1}^{n} p(X_i \mid pa(X_i))$$

Here pa(X_i) denotes a value assignment to the parents of X_i. (If an arrow in a graph is directed from A to B, then A is a parent node and B a child node).

The net result is that the probability distribution for a large set of variables may be represented by a product of the conditional probability relationships between small clusters of semantically related propositions. Now, instead of needing to elicit a joint probability distribution over a set of complex events, the problem is broken down into the assessment of these conditional probabilities as parameters of the graphical representation.

The lessons from this section can be summarised quite succinctly. First, Bayesian network graphs may be used to represent qualitative influences in a domain. Secondly, the conditional independence statements implied by the graph can be used to factorise the associated probability distribution. This factorisation can then be exploited to (a) ease the problem eliciting the global probability distribution, and (b) allow the development of computationally efficient algorithms for updating probabilities on the receipt of evidence. We will now describe how these techniques have been exploited to produce a BN model for software defect prediction.

3.3. Evidence Propagation in Bayesian Networks

Once a BN is built it can be executed using an appropriate propagation algorithm, such as the Hugin algorithm [Jensen 1996]. This involves calculating the joint probability table for the model (probability of all combined states for all nodes) by exploiting the BN's conditional probability structure to reduce the computational space. Even then, for large BNs that contain undirected cycles the computing power needed to calculate the joint probability table directly from the conditional probability tables is enormous. Instead, the junction tree representation is used to localise computations to those nodes in the graph that are directly related. The BN graph is transformed into the junction tree by collapsing connected nodes into cliques, eliminating cyclic links between cliques and by creating separators to communicate probability updates between the cliques when new evidence is observed. The key point here is that propagating the effects of observations throughout the BN can be done using only messages passed between – and local computations done within – the cliques of the junction tree rather than the full graph. The graph transformation process is computationally hard but it only needs to be produced once off-line. Propagation of the effects of new evidence in the BN is performed using Bayes' theorem over the compiled junction tree. For full details see [Jensen 1996].

Once a BN has been compiled it can be executed and exhibits the following two key features:

- The effects of observations entered into one or more nodes can be propagated throughout the net, in any direction, and the marginal distributions of all nodes updated;
- Only *relevant* inferences can be made in the BN. The BN uses the conditional dependency structure and the current knowledge base to determine which inferences are valid.

4. THE BAYESIAN NETWORK FOR DEFECT PREDICTION

We will now look at how Bayesian Networks (BNs) can be applied to quality prediction. The network we will describe was developed as a pilot study. However, the results that we obtained were quite positive – indeed rather more successful than we were expecting. The important point to mention at the outset is that the network was developed using experience form one part of a multinational organisation (Philips Electronics) in one continent (Europe). It was then validated using data from a different part of that organisation, in a different continent (India). Although these were two parts of the same organisation, it provides us with some confidence that the approach we are about to describe is capable of generalisable prediction models.

Assessing and controlling software quality is hard. You cannot hold it or touch it, yet its behaviour has an impact on all of our lives. We all are stakeholders in the drive to improve the quality of the software that we work with, yet few of us are able to explicate precisely how we define measures to discriminate between "poor" quality and "high" quality products.

This may seem strange as quality control is a precise science in most other industries, and an important product discriminator. There are, however, a number of reasons for this. Consider three main aspects of quality control in traditional manufacturing:

- The control of manufacturing defects
- The assessment of mean time to failure of a product through wear or ageing
- The use of statistical sampling to provide quality predictions with well-defined uncertainties

In general, these have limited applicability in software engineering. The main reason for this is that in software engineering we are concerned with controlling the *design* process and not the *manufacturing* process. We want to:

- Know how to control the design and development process so that design faults and weaknesses are minimised
- Assess the likelihood that failures to meet the quality requirements of users (through design and development faults) will be manifest in a specific context of use – and, ideally, how that likelihood might vary as the context of use (inevitably) changes over time
- Develop quality measurement and assessment techniques that can be applied in cases where a specific design and development process may only be applied to a small number of projects – perhaps even just an individual project.

The model we describe focuses on one specific quality characteristic – what we may call *maturity*, or freedom from defects. We will construct a model which we hypothesise contains the most important casusal influences on the presence of defects in a software module. Note that the model focuses specifically on functionality related defects, and not faults in performance or other quality requirements. The latter will be addressed in the next phase of this research programme.

BN models are a good candidate solution for an effective model of software defect prediction for the following reasons:

1. They can easily model causal influences between variables in a specified domain;
2. The Bayesian approach enables statistical inference to be augmented by expert judgement in those areas of a problem domain where empirical data is sparse;
3. As a result of the above, it is possible to include variables in a software reliability model that correspond to process as well as product attributes;
4. Assigning probabilities to reliability predictions means that sound decision-making approaches using classical decision theory can be supported.

Our goal was to build a module level defect prediction model that could then be evaluated against real project data. Although it was not possible to use members of Philips' development organisations directly to perform extensive knowledge elicitation, PRL were able to act as a surrogate because of their experience from working directly with Philips business units. This had the added advantage that the BN could be built relatively quickly. However, the fact that the probability tables were in effect built from "rough" information sources and strengths of relations necessarily limits the precision of the model.

The remainder of this section will provide an overview of the model to indicate the product and process factors that are taken into account when a quality assessment is performed using it.

4.1. Overall structure of the Bayesian Network

The BN is executed using the generic probabilistic inference engine Hugin (see http://www.hugin.com for further details). However, the size and complexity of the network were such that it was not realistic to attempt to build the network directly using the Hugin tool. Instead, Agena Ltd used two methods and tools that are built on top of the Hugin propagation engine:

- The SERENE method and tool [Fenton, 1999], which enables: large networks to be built up from smaller ones in a modular fashion; and, large probability tables to be built using pre-defined mathematical functions and probability distributions.
- The IMPRESS method and tool [Neil, 1999], which extends the SERENE tool by enabling users to generate complex probability distributions simply by drawing distribution shapes in a visual editor.

The resulting network takes account of a range of product and process factors from throughout the lifecycle of a software module. Because of the size of the model, it is impractical to display it in a single figure. Instead, we provide a first schematic view in terms of sub-nets (Figure 4.1). This modular structure is the actual decomposition that was used to build the network using the SERENE tool.

The main sub-networks (sub-nets) in the high-level structure correspond to key software life-cycle phases in the development of a software module. Thus there are sub-nets representing the specification phase, the specification review phase, the design and coding phase and the various testing phases. Two further sub-nets cover the influence of requirements management on defect levels, and operational usage on defect discovery. The final defect density sub-net simply computes the industry standard defect density metric in terms of residual defects delivered divided by module size.

This structure was developed using the software development processes from a number of Philips development units as models. A common software development process is not currently in place within Philips. Hence the resulting structure is necessarily an abstraction. Again, this will limit the precision of the resulting predictions. Work is in progress to develop tools to enable the structure to be customised to specific development processes.

The arc labels in Figure 4.1 represent 'joined' nodes in the underlying sub-nets. This means that information about the variables representing these joined nodes is passed directly between sub-nets. For example, the *specfication quality* and the *defect density* sub-nets are joined by an arc labelled 'Module size'. This node is common to both sub-nets. As a result, information about the module size arising from the specification quality sub-net is passed directly to the defect density sub-net. We refer to 'Module size' as an 'output node' for the *specification quality* sub-net, and an 'input node' for the *defect density* sub-net. The figures in the following sub-sections show details of a number of sub-nets. In these figures, the dark shaded nodes with dotted edges are output nodes, and the dark shaded ones with solid edges are input nodes.

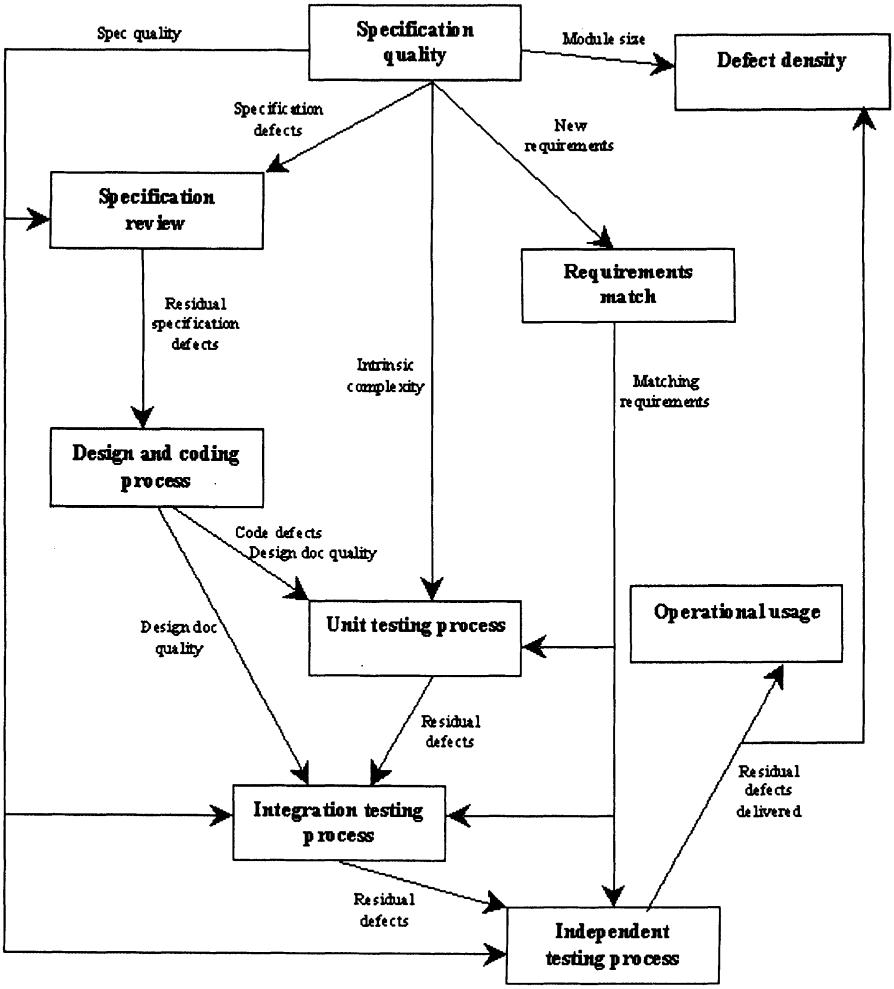


Figure 4.1. Overall network structure.

4.2. The specification quality sub-net

Figure 4.2 illustrates the Specification quality sub-net. It can be explained in the following way: *specification quality* is influenced by three major factors:

- the *intrinsic complexity* of the module (this is the complexity of the requirements for the module, which ranges from "very simple" to "very complex");
- the *internal resources* used, which is in turn defined in terms of the *staff quality* (ranging from "poor" to "outstanding"), the *document quality* (meaning the quality of the initial requirements specification document, ranging from "very poor" to "very good"), and the *schedule* constraints which ranges from "very tight" to "very flexible";
- the *stability* of the requirements, which in turn is defined in terms of the *novelty* of the module requirements (ranging from "very high" to "very low") and the *stakeholder involvement* (ranging from "very low" to "very high"). The stability node is defined in such a way that low novelty makes stakeholder involvement irrelevant (Philips would have already built a similar relevant module), but otherwise stakeholder involvement is crucial.

The *specification quality* directly influences the number of *specification defects* (which is an output node with an ordinal scale that ranges from 0 to 10 – here "0" represents no defects, whilst "10" represents a complete rewrite of the document). Also, together with *stability*, specification quality influences the number of *new requirements* (also an output node with an ordinal scale ranging from 0 to 10) that will be introduced during the development and testing process. The other node in this sub-net is the output node *module size*, measured in Lines of Code (LOC). The position taken when constructing the model is that module size is conditionally dependent on *intrinsic complexity* (hence the link). However, although it is an indicator of such complexity the relationship is fairly weak - the Node Probability Table (NPT) for this node models a shallow distribution.

4.3. The Requirements match sub-net

The Requirements match sub-net (Figure 4.3) contains just three nodes. These could have been incorporated into the specification quality sub-net, but we have separated them out as a sub-net to highlight the overall importance that we attach to the notion of *requirements match.* This crucial output variable (ranging from poor to very good) represents the extent to which the implementation matches the real requirements. It is influenced by the number of *new requirements* and the quality of *configuration and* *traceability management*. When there are new requirements introduced, if the quality of configuration and traceability management is poor then it is likely that the requirements match will be poor. This will have a negative impact on all subsequent testing phases (hence this node is input to three other sub-nets that model testing phases). For example, if the requirements match is poor then no matter how good the internal development is, when it comes to the integration and independent testing phases the testers will inevitably be testing the wrong requirements.

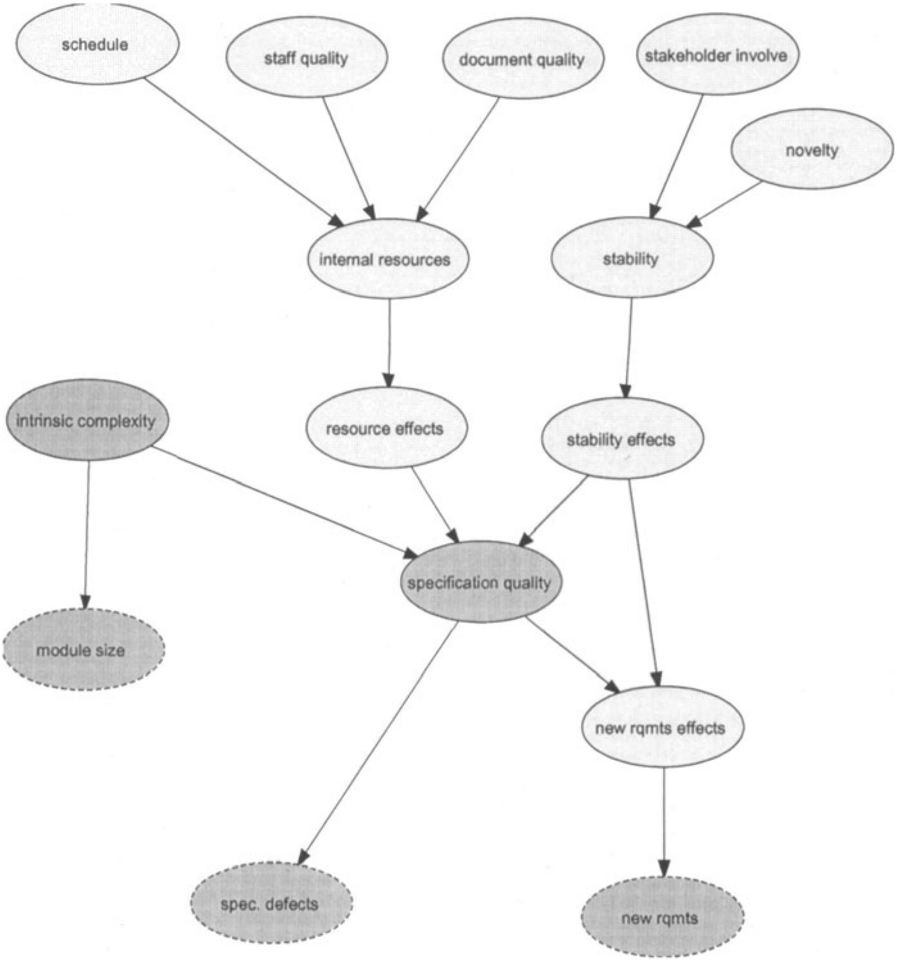


Figure 4.2. Specification quality sub-net.

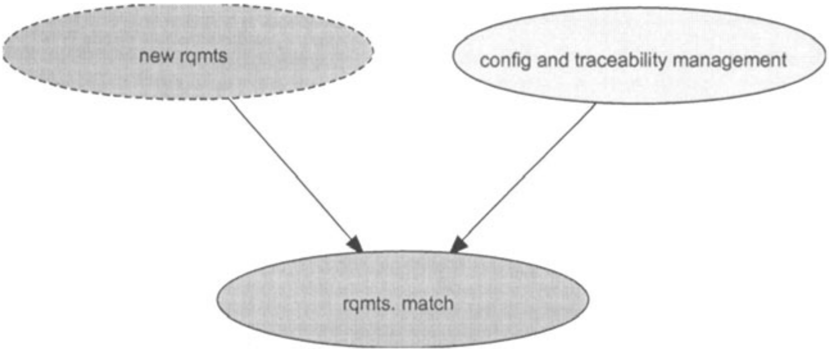


Figure 4.3. Requirements match sub-net.

4.4. The Specification Review and Test Process sub-nets

The Specification Review, Unit, Integration and Independent testing process, and Operational usage sub-nets are all based on a common testing idiom (Figure 4.4). The basic structure of each is that they receive defects from the previous life-cycle phase as ‘inputs’, and the accuracy of testing and rework is dependent on the resources available. The ‘output’ in each case is the unknown number of residual defects, which is simply the number of inserted defects minus the number of discovered defects.

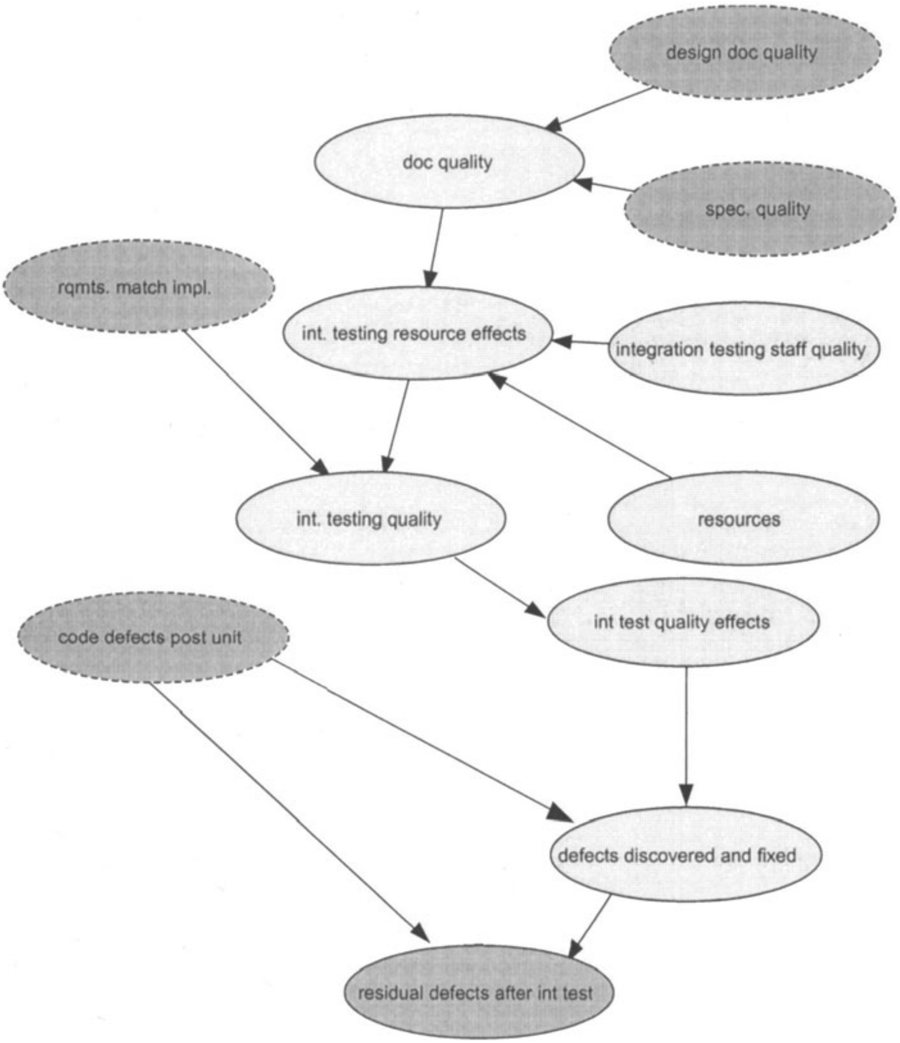


Figure 4.4. Integration testing process sub-net. This is an example of the generic testing idiom.

4.5. Design and coding process sub-net

The Design and coding process sub-net (Figure 4.5) is an example of the so-called "process-product" idiom. Based on various input resources something is being produced (namely design and code) that has certain attributes (which are the outputs of the sub-net). The inputs here are *specification quality* (from the specification quality sub-net), *development staff quality* and *resources*. These three variables define the *design and* *coding quality*. The output attributes of the design and coding process are the *design document quality* and the crucial number of *code defects introduced*. The latter is influenced not just by the quality of the design and coding process but also by the number of residual specification defects (an input from the specification sub-net).

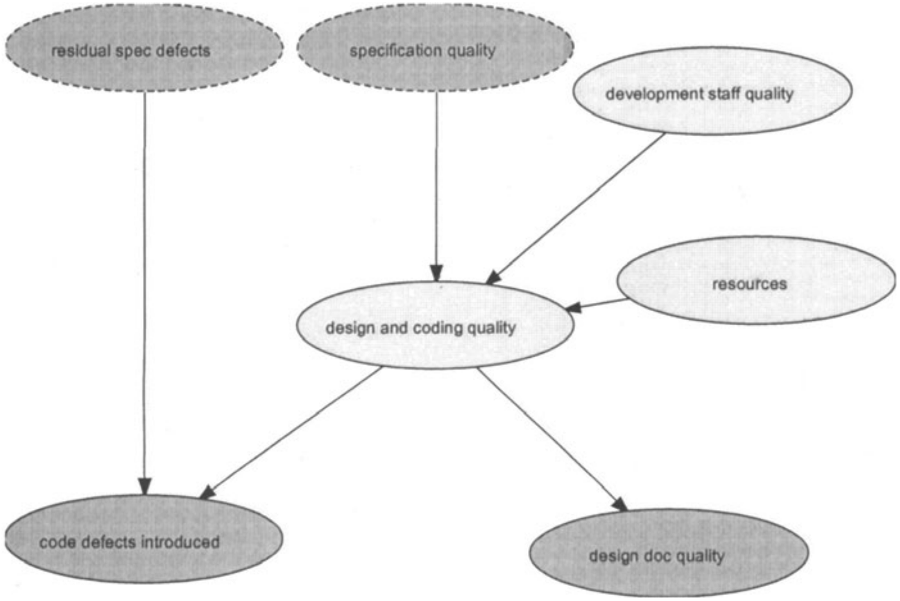


Figure 4.5. Design and coding process sub-net — an example of the "process-product" idiom.

4.6. Defect density sub-net

The final sub-net is the Defect density sub-net (Figure 4.6). This sub-net simply computes the industry standard defect density metric in terms of residual defects delivered divided by module size. Notice that *defect density* is an example of a node that is related to its parents by a deterministic, as opposed to a probabilistic, relationship. This ability to incorporate deterministic nodes was an important contribution of the SERENE project.

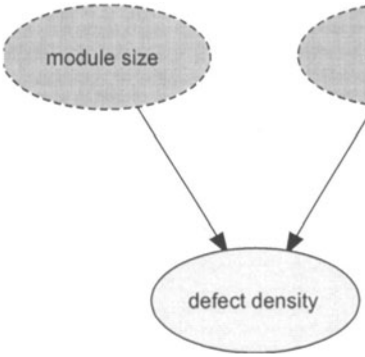


Figure 4.6. The Defect density sub-net.

4.7. The probability tables

The work on BNs outlined in Section 3 means that the problem of building such models now factorises into two stages:

- *Qualitative stage*: consider the general relationships between the variables of interest in terms of relevance of one variable to another in specified circumstances;
- *Quantitative stage*: numerical specification of the parameters of the model.

The numerical specification of the parameters means building Node Probability Tables (NPTs) for each of the nodes in the network. However, although the problem of eliciting tables on a node-by-node basis is cognitively easier than eliciting global distributions, the sheer number of parameters to be elicited remains a very serious handicap to the successful building of BNs. We will outline some of the techniques we used to handle this problem in this sub-section.

Note that for reasons of commercial sensitivity, the parameter values used in this paper may not correspond to the actual values used.

The leaf nodes (those with no parents) are the easiest to deal with since we can elicit the associated marginal probabilities from the expert simply by asking about frequencies of the individual states. For example, consider the leaf node *novelty* in Figure 4.2. This node has five states "very high", "high", "average", "low", "very low". Suppose the expert judgement is that modules typically are not very novel, giving the following weights (as surrogates for the probability distribution), respectively, on the basis of knowledge of all previous modules in a development organisation:

5,10,20,40,20

These are turned into probabilities 0.05, 0.11, 0.21, 0.42, 0.21 (note the slight change of scale to normalise the distribution).

The NPTs for all other leaf nodes were determined in a similar manner (by either eliciting weightings or a drawing of the shape of the marginal distribution).

The NPTs for nodes with parents are much more difficult to define because, for each possible value that the node can take, we have to provide the conditional probability for that value with respect to every possible combination of values for the parent nodes. In general this cannot be done by eliciting each individual probability – there are just too many of them (there are several million in total in this BBN). Hence we used a variety of methods and tools that we have developed in recent projects SERENE [1999] and IMPRESS [1999]. For example, consider the node *specification quality* in Figure 4.2. This has three parent nodes *resources*, *intrinsic complexity*, and *stability* each of which takes on several values (the former two have 5 values and the latter has 4). Thus for each value for specification quality we have to define 100 probabilities. Instead of eliciting these all directly we elicit a sample, including those at the 'extreme' values as well as typical, and ask the expert to provide the rough shape of the distribution for specification quality in each case. We then generate an actual probability distribution in each case and extrapolate distributions for all the intermediate values. To see how this was done, Table 1 shows the actual data we elicited in this case. The first three columns represent the specific sample values and the final column is the rough shape for the distribution of "specification quality" given those values.

In the "best case" scenario of row 2 (resources good, stability high, complexity low) the distribution peaks sharply close to 5 (i.e. close to "best" quality specification). If the complexity is high (row 3) then the distribution is still skewed toward the best end, but is not as sharply peaked. In the "worst case" scenario of row 3 (resources bad, stability low, complexity high) the distribution peaks sharply close to 1 (i.e. close to "worst" quality specification).

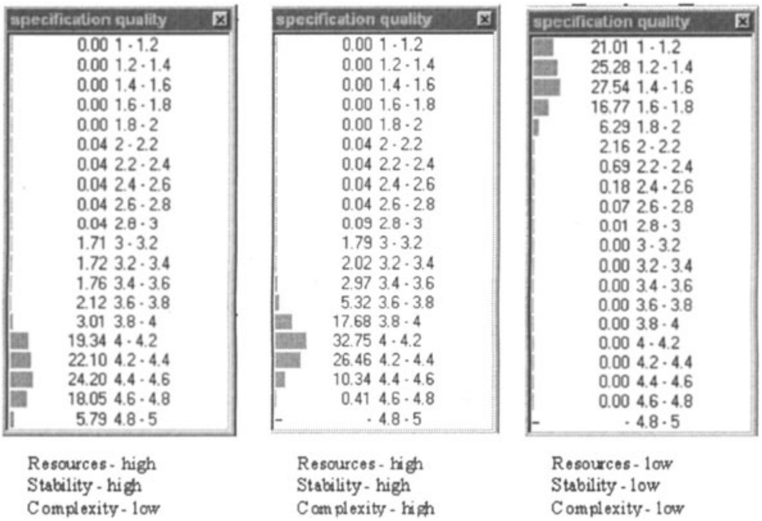
On the basis of the distributions drawn by the expert we derive a function to compute the mean of the specification quality distribution in terms of the parents variables. For example, in this case the mean used was:

Min (resource_effects, (5*resource_effects+intrinsic_complexity+5
*stability_effects) / 11)

In this example, to arrive at the distribution shapes drawn by the expert, we make use of intermediate nodes as described in Section 4.2. For example, there is an intermediate node *stability* which is the parent of the node *stability effects*. The *stability effects* node NPT is defined as the following beta distribution that is generated using the IMPRESS tool:

Beta (2.25 * stability - 1.25, -2.25 * stability + 12.25, 1, 5)

Figure 4.7 shows the actual distribution in the final BBN (using the Hugin tool) for the node specification quality under a number of the scenarios of Table 1. This figure provides a good consistency check – there is an excellent match of the distributions with those specified by the expert.



specification quality

0.00	1 - 1.2
0.00	1.2 - 1.4
0.00	1.4 - 1.6
0.00	1.6 - 1.8
0.00	1.8 - 2
0.04	2 - 2.2
0.04	2.2 - 2.4
0.04	2.4 - 2.6
0.04	2.6 - 2.8
0.04	2.8 - 3
1.71	3 - 3.2
1.72	3.2 - 3.4
1.76	3.4 - 3.6
2.12	3.6 - 3.8
3.01	3.8 - 4
19.34	4 - 4.2
22.10	4.2 - 4.4
24.20	4.4 - 4.6
18.05	4.6 - 4.8
5.79	4.8 - 5

Resources - high
Stability - high
Complexity - low

specification quality

0.00	1 - 1.2
0.00	1.2 - 1.4
0.00	1.4 - 1.6
0.00	1.6 - 1.8
0.00	1.8 - 2
0.04	2 - 2.2
0.04	2.2 - 2.4
0.04	2.4 - 2.6
0.04	2.6 - 2.8
0.09	2.8 - 3
1.79	3 - 3.2
2.02	3.2 - 3.4
2.97	3.4 - 3.6
5.32	3.6 - 3.8
17.68	3.8 - 4
32.75	4 - 4.2
26.46	4.2 - 4.4
10.34	4.4 - 4.6
0.41	4.6 - 4.8
-	4.8 - 5

Resources - high
Stability - high
Complexity - high

specification quality

21.01	1 - 1.2
25.28	1.2 - 1.4
27.54	1.4 - 1.6
16.77	1.6 - 1.8
6.29	1.8 - 2
2.16	2 - 2.2
0.69	2.2 - 2.4
0.18	2.4 - 2.6
0.07	2.6 - 2.8
0.01	2.8 - 3
0.00	3 - 3.2
0.00	3.2 - 3.4
0.00	3.4 - 3.6
0.00	3.6 - 3.8
0.00	3.8 - 4
0.00	4 - 4.2
0.00	4.2 - 4.4
0.00	4.4 - 4.6
0.00	4.6 - 4.8
-	4.8 - 5

Resources - low
Stability - low
Complexity - low

Figure 4.7. Actual distributions for *specification quality* for various scenarios.

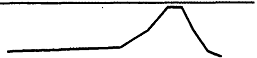
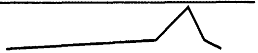
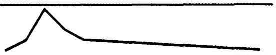
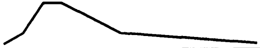
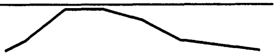
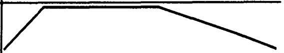
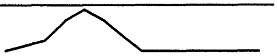
Resources (1 to 5 where 1 is worst 5 is best)	Stability (1 to 3 where 1 is worst 3 is best)	Intrinsic complexity (1 to 5 where 1 is most complex, 5 least)	Specification quality 1 2 3 4 5
5	3	1	
5	3	5	
1	1	1	
1	2	3	
1	3	5	
5	1	1	
1	1	5	

Table 1. Eliciting the probability table for *specification quality*.

4.8. Some comments on the BN

The methods used to construct the model have been illustrated in this section. The resulting network models the entire development and testing life-cycle of a typical software module. We believe it contains all the critical causal factors at an appropriate level of granularity, at least within the context of software development within Philips for single site, single team projects.

The node probability tables (NPTs) were built by eliciting probability distributions based on experience from within Philips. Some of these were based on historical records, others on subjective judgements. For most of the non-leaf nodes of the network the NPTs were too large to elicit all of the relevant probability distributions using expert judgement. Hence we used the novel techniques, that have been developed recently on the SERENE and IMPRESS projects, to extrapolate all the distributions based on a small number of samples. By applying numerous consistency checks we believe that the resulting NPTs are a fair representation of experience within Philips.

There are two major concerns with the model as it stands. We address the first to an extent in developing the network into a tool that can be used for validation studies. However, the second can only be addressed as further experience is gained with real useage of the tool.

The first concerns the measurement scales on many of the nodes. One or two of the nodes have objective measurement scales (such as lines of code for “module size”, although even here one needs to be precise about how this is measured). However, an ordinal scale of 1 (worst case) to 5 (best case) is more often used. This leaves scope for subjective judgement of the values of these nodes in any particular project. We handled this in the user interface that was developed for the validation studies by providing definitions for each of the values of a measurement scale. Consider for example, the node for configuration and traceability management. Here we define the worst case state as “requirements and design documents are no longer maintained once coding has started”, the best case as “requirements, design and code are maintained with tool support for establishing and maintaining traceability links”. We do need to study further the validity of these categories and their respective orderings, and the scope for intra- and inter-subject variability in the value assignments. So, and this is important, we cannot claim this as a definitive model. Rather, the success of the validation studies indicate that it is a good foundation for further evolution.

This leads us on to the second point. The model captures a (limited) corpus of experience in software engineering. To the extent that we talk about “causal influences”, we are *hypothesising* a “theory” of software engineering (in fact, of a limited sub-domain of software engineering). To that extent, we must be prepared to revise it and extend it as our experience matures. A key motive for building our model as a Bayesian Network is that it *is* revisable, both in terms of its structure and its parameters. Rather then try to learn a model purely from data in a domain where the number of potential influences is vast, and the data from controlled evaluations of the effects of each influence is minimal, we start with a plausible model that is set up so that it can be revised as further data and experience is gained. This, for example, is a motivation behind the specific use of beta-distributions in the NPTs. These are one of a class of conjugate distributions. That is, the distribution posterior to revision by data has the same form as prior to revision. In introduction to learning Bayesian Networks can be found in [Krause, 1998] which has extensive references to the machine learning techniques that are now open to us.

As it stands, the BN can be used to provide a range of predictions and "what-if" analyses at any stage during software development and testing. It can be used both for quality control and process improvement. However, two further areas of work were needed before the tool could be considered ready for extended trials. Firstly and most importantly, the network needed to be validated using real-world data. Secondly a more user-friendly interface needed to be engineered so that (a) the tool did not require users to have experience with BNs, and (b) a wider range of reporting functions could be provided. The validation exercise will be described in the next section in a way that illustrates how the probabilistic network was packaged to form the AID tool (AID for "Assess, Improve, Decide").

5. VALIDATION OF THE AID TOOL

5.1. Method

The Philips Software Centre (PSC), Bangalore, India, made validation data available. We gratefully acknowledge their support in this way. PSC is a centre for excellence for software development within Philips, and so data was available from a wide diversity of projects from the various Business Divisions within PSC.

Data was collected from 28 projects from three Business Divisions: Mainstream Consumer Electronics, Philips Medical Systems and Digital Networks. This gave a spread of different sizes and types of projects. Data was collected from three sources:

- Pre-release and post-release defect data was collected from the "Performance Indicators" database.

- More extensive project data was available from the Project Database.
- Completed questionnaires on selected projects.

In addition, the network was demonstrated in detail on a one to one basis to three experienced quality/test engineers to obtain their reaction to its behaviour under a number of hypothetical scenarios.

The data from each project was entered into the BN model. For each project:

1. The data available for all nodes prior to the Unit Test sub-net was entered first.
2. Available data for the Unit Test sub-net was then entered, with the exception of data for defects discovered and fixed.
3. If pre-release defect data was available, the predicted probability distribution for defects detected and fixed in the unit test phase was compared with the actual number of pre-release defects. No distinction was made between major and minor defects – total numbers were used throughout. The actual value for pre-release defects was then entered.
4. All further data for the test phases was then entered where available, with the exception of the number of defects found and fixed during independent testing ("post-release defects"). The predicted probability distribution for defects found and fixed in independent testing was compared with the actual value.
5. If available, the actual value for the number of defects found and fixed during independent testing was then entered. The prediction for the number of residual defects was then noted.

Unfortunately, data was not available to validate the operational usage sub-net. This will need data on field call-rates that is not currently available.

Given the size of the BN, this was insufficient data to perform rigorous statistical tests of validity. However, it was sufficient data to be able to confirm whether or not the network's predictions were reliable enough to warrant recommending that a more extensive controlled trial be set up.

5.2. Initial Validation Results

The network was used to make predictions of numbers of defects found during unit test, integration test and independent testing of the module once it had been integrated into a product. These predictions were compared against the actual values obtained. Data from an initial validation run is presented in Table 2.

With the exception of three predictions, there is good agreement between the predictions and the actual defect numbers when available. Note, however, that:

1. The sample size is too small to use any statistical measures of validity;
2. The AID predictions are quite imprecise, deliberately so at this stage in its evolution, and so is not capable of drawing particularly fine distinctions between projects. In particular, there was little discrimination between the quality of the projects once they had been released to independent test, although this may also be a reflection of the repeatability of the development processes at PSC.

Project ID	Unit Test		Independent Test	
	Predicted defects	Actual defects	Predicted defects	Actual defects
P1	100-150	122	20-40	31
P2	100-150	141	40-60	NA
P3	40-60	7	20-40	22
P4	60-80	11	20-40	46
P5	100-150	142	20-40	NA
P6	100-150	370	20-40	NA

Table 2. Comparison of predicted versus actual defects for six projects.

Projects P3 and P4 showed significant differences between the predicted and actual defects discovered, or at least, reported, during unit test. This reflects a possible difficulty with use of the tool in practice, since many defects at unit test are fixed without going through a formal reporting process.

The largest discrepancy was, however, with the project P6. This project contained a significant User Interface component, and as validation progresses we continue to see that such projects do consistently produce more incident reports than are predicted by AID. The explanation for this is not immediately apparent, and clearly the causes are not currently captured in the AID tool.

5.3. Emergent behaviour from complex modules

One of the major values of AID is as a tool for exploring the possible consequences of changes to a software process, or the constraints on a product's development. The ability of Bayesian networks to handle quite complex reasoning patterns is one of the reasons why the tool is proving so successful in this regard. We end with one example, which also illustrated how our model does handle the sort of effects that were discussed in Section 2.

Table 3 lists the median values of “Defects found at Unit Test” and “Defects Delivered” for a variety of values for the intrinsic problem complexity of the software module under development. Look at the first row; the predictions for the number of defects found during unit test. For a very simple module, we get an increase in the number of defects found over the prior, and a decrease for a very complex module.

	Intrinsic Complexity of the Software Module		
	Prior	“Very Simple”	“Very Complex”
Defects found in Unit Test	90	125	30
Defects delivered	50	30	70

Table 3. Median values for three different scenarios.

At first sight, this seems counter intuitive – we might expect simpler modules to be more reliable. The explanation is that the more complex modules are harder to test than the simpler modules. With their greater ability to “hide” faults, fewer faults will be detected unless there is a compensating increase in the effectiveness with which the module is tested. No such compensation has been applied in this case and the low prediction for defects detected and fixed for the “very complex” case indicates that typically such modules are relatively poorly tested.

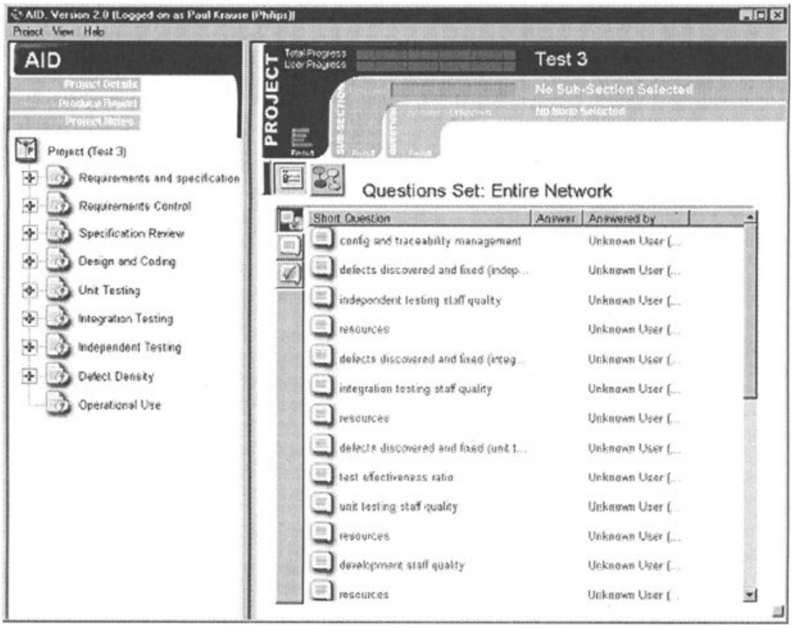
This is borne out when we look at the respective figures for residual defects delivered, in the second row of the table. Now we see a reversal. The prediction for the “very complex” module indicates that it will contain more residual defects than the “very simple” module (a median of 70, compared to a median of 30). So our model naturally produces the qualitative behaviour of the real world data from our earlier experiment. That is, the better-tested modules yield more defects during unit test and deliver fewer defects. For the more poorly tested modules, the converse is the case. (Note that the table misses out data from the Integration and Independent Test Phases. When this is included the total number of defects – found plus delivered – is greatest for the “Very Complex” module).

5.4. An example run of AID

We will use screen shots of the AID Tool to illustrate both the questionnaire based user interface, and a typical validation run.

One of the concerns with the original network is that many of the nodes have values on a simple ordinal scale, range from "very good" to "very poor". This leaves open the possibility that different users will apply different calibrations to these scales. Hence the reliability of the predictions may vary, dependent on the specific user of the system. We address this by providing a questionnaire based front-end for the system. The ordinal values are then associated with specific question answers. The answers themselves are phrased as categorical, non-judgemental statements.

The screen in Figure 5.1 shows the entire network. The network is modularised so that a Windows Explorer style view can be used to navigate quickly around the network. Check-boxes are provided to indicate which questions have already been answered for a specific project.

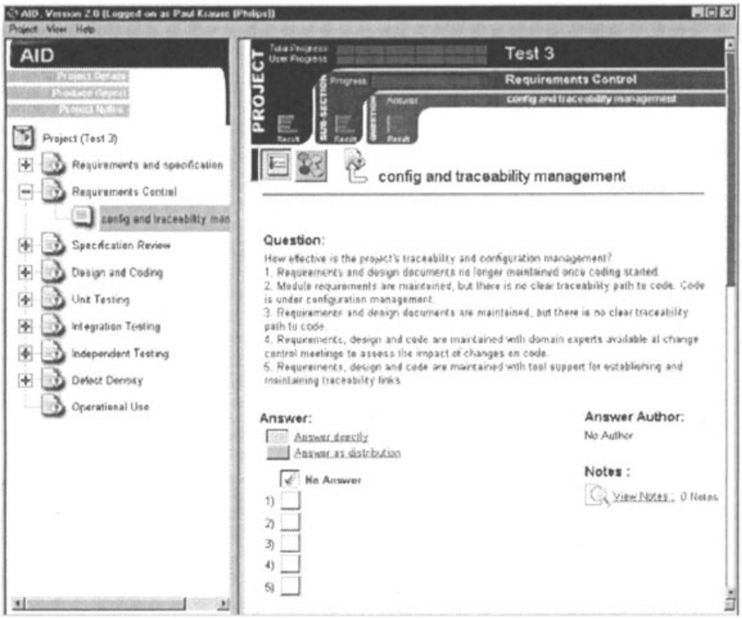


Short Question	Answer	Answered by
config and traceabilty management		Unknown User (...
defects discovered and fixed (indep...		Unknown User (...
independent testing staff quality		Unknown User (...
resources		Unknown User (...
defects discovered and fixed (integ...		Unknown User (...
integration testing staff quality		Unknown User (...
resources		Unknown User (...
defects discovered and fixed (unit t...		Unknown User (...
test effectiveness ratio		Unknown User (...
unit testing staff quality		Unknown User (...
resources		Unknown User (...
development staff quality		Unknown User (...
resources		Unknown User (...

Figure 5.1. The entire AID network illustrated using a Windows Explorer style view.

The questions associated with a specific sub-net can then be displayed. A question is answered by selecting the alternative from the suggested answers that best matches the state of current project. Figure 5.2 shows the question and alternative answers for the Configuration and Traceability Management node in the Requirements Control sub-network.

For this example project, answers were available for 13 of the 16 questions preceding "defects discovered and fixed during unit test". Once the answers to these questions were entered, the predicted probability distribution for defects discovered and fixed during unit test had a mean of 149 and median of 125 (See Figure 5.3 – in this figure the monitor window has been displayed in order to show the complete probability distribution for this prediction. Summary statistics can also be displayed.). The actual value was 122. Given that the probability distribution is skewed, the median is the most appropriate summary statistic, so we actually see an apparently very close agreement between predicted and actual values. This agreement was very surprising as although we were optimistic that the "qualitative behaviour" of the network to be transferable from organisation to organisation, we were expecting the scaling of the defect numbers to vary. Note, however, that the median is an imprecise estimate of the number of defects – it is the centre value of its associated bin on the histogram. So it might be more appropriate to quote a median of "100-150" in order to make the imprecision of the estimate explicit.



Question:
How effective is the project's traceability and configuration management?
1. Requirements and design documents no longer maintained once coding started.
2. Module requirements are maintained, but there is no clear traceability path to code. Code is under configuration management.
3. Requirements and design documents are maintained, but there is no clear traceability path to code.
4. Requirements, design and code are maintained with domain experts available at change control meetings to assess the impact of changes on code.
5. Requirements, design and code are maintained with tool support for establishing and maintaining traceability links.

Answer:
Answer directly
Answer as distribution
☑ No Answer
1) □
2) □
3) □
4) □
5) □

Answer Author:
No Author

Notes :
View Notes : 0 Notes

Figure 5.2. The question associated with the Configuration and Traceability Management node.

The actual value for defects discovered and fixed was entered. Answers for "staff quality" and "resources" were available for the Integration Test and Independent Test sub-networks. Once these had been entered, the prediction for defects discovered and fixed during independent test had a mean of 51, median of 30 and standard deviation of 45 (see figure 5.4). The actual value was 31.

As was the case with unit test, there was close agreement between the median of the prediction and the actual value. "Test 3" was developed by PSC as a module or sub-system for a specific Philips development group. The latter then integrated "Test 3" into their product, and tested the complete product. This is the test phase we refer to as Independent Test.

The code size of Test 3 was 144 KLOC. The modules (perhaps sub-system is a better term given the size) used in the validation study ranged in size from 40-150 KLOC. The probabilistic reliability model incorporates a relatively weak coupling between module size and numbers of defects. The results of the validation continue to support the view that other product and process factors have a more significant impact on numbers of defects. However, we did make one modification to the specification quality sub-net as a result of the experience gained during the validation. Instead of "Intrinsic Complexity" being the sole direct influence on "Module Size", we have now explicitly factored out "Problem Size" as a joint influence with "Intrinsic Complexity" on "Module Size".

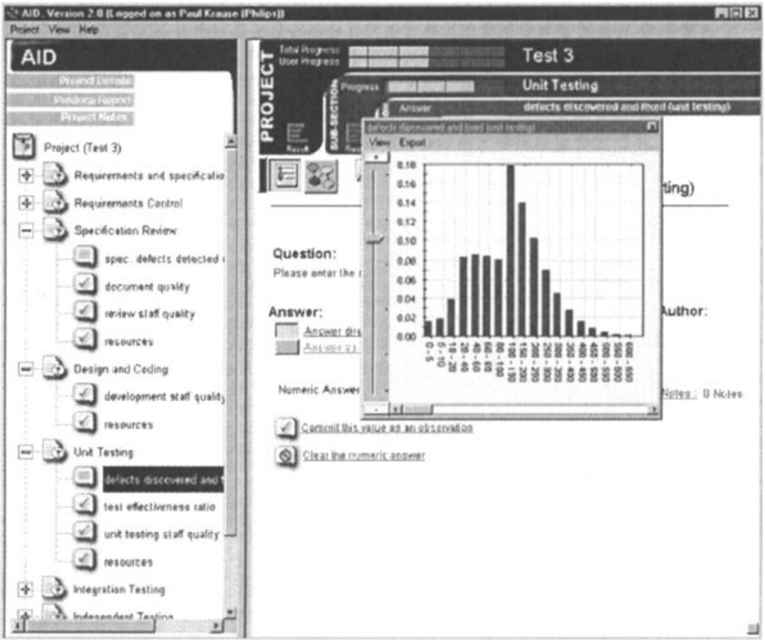


Figure 5.3. The prediction for defects discovered and fixed during Unit Test for project "Test 3".

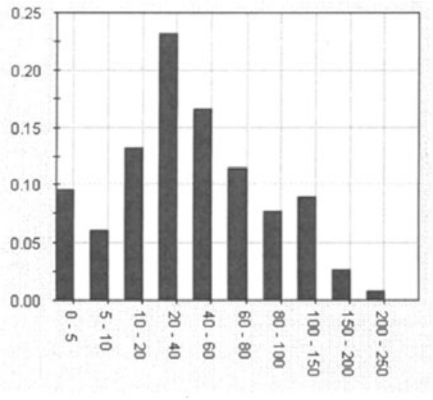


Figure 5.4. The prediction for defects discovered and fixed during Independent Test for project "Test 3".

5.5. Summary of results of the validation exercise

Overall there was a high degree of consistency between the behaviour of the network and the data that was collected. However, a significant amount of data is needed in order to make reasonably precise predictions for a specific project. Extensive data (filled questionnaire, plus project data, plus defect data) was available for seven of the 28 projects. These seven projects showed a similar degree of consistency to the project that will be studied in the next sub-section. The remaining 21 projects show similar effects, but as the probability distributions are broader (and hence less precise) given the significant amounts of "missing" information, the results are supportive but less convincing than the seven studied in detail.

It must be emphasised that all defect data refers to the total of major and minor defects. Hence, residual defects may not result in a "failure" that is perceptible to a user. This is particularly the case for user-interface projects.

Note also that the detailed contents of the questionnaires are held in confidence. Hence we cannot publish an example of data entry for the early phases in the software life cycle. Defect data is reported here, but we must keep the details of the project anonymous.

A disadvantage of a reliability model of this complexity is the amount of data that is needed to support a statistically significant validation study. As the metrics programme at PSC is relatively young (as is the organisation itself), this amount of data was not available. As a result, we were only able to carry out a less formal validation study. Nevertheless, the outcome of this study was very positive. Feedback was obtained on various aspects of the functionality provided by the AID interface to the reliability model, yet the results indicated that only minor changes were needed to the underlying model itself. We are now preparing for a more extended trial using a wider range of projects.

There is a limit to what we can realistically expect to achieve in the way of statistical validation. This is inherent in the nature of software engineering. Even if a development organisation conforms to well defined processes, they will not produce homogenous products – each project will differ to an extent. Neither do we have the large relevant sample sizes necessary for statistical process control. It is primarily for these reasons that we augment empirical evidence with expert judgement using the Bayesian framework described in this paper. As more data becomes available, it is possible to critique and revise the model so that the probability tables move from being subjective estimates to being a statement of physical properties of the world (see, e.g. [Krause, 1998]). However, in the absence of an extensive and expensive reliability testing phase, this model can be used to provide an estimate of residual defects that is sufficiently precise for many software project decisions.

5.6. Future Work

The scale of new software systems is such that it is increasingly necessary to develop them across *distributed* environments. Philips, for example, has teams on three different continents all working on the same major projects in a number of instances. Such distributed projects raise new and urgent concerns about how to manage and monitor quality and risks. To address this important concern Agena, QinetiQ, Philips and the Israel Aircraft Industry are currently working on a 2.4m Euro project – MODIST [Elliot 2001] to develop a BN-based tool to support software managers and engineers improve their software development processes as well their product quality in distributed environments.

6. SUMMARY

We have described a BN for software defect prediction. This model can be used for assessing ongoing projects, but also for exploring the possible effects of a range of software process improvement activities. If costs can be associated with process improvements, and benefits assessed for the predicted improvement in software quality, then the model can be used to support sound decision making for SPI (Software Process Improvement).

The model performed very well in our preliminary validation experiments. In addition, Agena developed a user interface for the tool that enables it to be easily used in a variety of different modes for product assessment and SPI.

We must emphasise that this is an initial model. As discussed in Section 4, it should be viewed as a hypothesis representing a "theory" of software development in a sub-domain of software engineering (single-roof, single team software development of software modules). The use of a Bayesian approach provides us with an initial model in a domain that is notoriously sparse in sound empirical data. As continued experience is gained, the model can be revised using sound statistical techniques. Indeed, such a model could provide a valuable basis for the design of experiments in empirical software engineering (by for example, controlling certain nodes in the network and predicting outcomes on the result nodes).

Although we anticipate that the model will need additional refinement as experience is gained during extended trials, we are confident that it will make a significant contribution to sound and effective decision-making in software development organisations. Future refinements to the method and the approach are being done on the CEC ESPRIT funded MODIST project [Elliot 2001].

ACKNOWLEDGEMENTS

Simon Forey carried out the user interface development for AID; his contribution to this project was invaluable. We would also like to thank Mr. S. Nagarajan, Mr. Thangamani N., Mr. Soumitra Lahiri, Mr. Jitendra Shreemali and all the others at PSC, Bangalore who helped in the validation study. The validation study was started with Mainstream CE projects at PSC, with valuable support from the Software Quality Engineering Team in this division.

REFERENCES

Adams E. (1984), "Optimizing preventive service of software products", IBM Research Journal, 28(1), 2-14.

Agena (2002) Agena Ltd, "Bayesian Belief Nets", http://www.agena.co.uk

Basili V., Briand L. and Melo W.L. (1996), "A validation of object oriented design metrics as quality indicators", IEEE Trans. Software Eng.

Cartwright M. and Shepperd M. (1997), "Building predictive models from object-oriented metrics," presented at 8th European Software Control and Metrics Conf., Berlin.

Elliot J. (2001). ESPRIT Project MODIST: Models of Uncertainty and Risk for Distributed Software Development. http://www.modist.org.uk/

Fenton N. (1999) SERENE consortium, "SERENE (SafEty and Risk Evaluation using Bayesian Nets): Method Manual", ESPRIT Project 22187, http://www.dcs.qmw.ac.uk/~norman /serene.htm.

Fenton N. and M. Neil (1999) "A Critique of Software Defect Prediction Research", IEEE Trans. Software Eng., 25, No.5, 675-689.

Fenton N. and N. Ohlsson (2000) "Quantitative analysis of faults and failures in a complex software system", IEEE Trans. Software Eng., 26, 797-814.

Fenton N.E. and S.L. Pfleeger (1997), Software Metrics: A Rigorous and Practical Approach, (2nd Edition), PWS Publishing Company.

Hugin (1998). Hugin Expert Brochure, Hugin Expert A/S, P.O. Box 8201 DK-9220 Aalborg, Denmark.

Khoshgoftaar T.M. and Munson J.C. (1990), "Predicting software development errors using complexity metrics". IEEE J of Selected Areas in Communications, Vol.8, No.2, pp.253-261, 1990.

Krause P.J. (1998) "Learning Probabilistic Networks", Knowledge Engineering Review, 13, 321-351.

Lauritzen S.L. and D.J. Spiegelhalter, (1988) "Local computations with probabilities on graphical structures and their application to expert systems (with discussion)" J. Roy. Stat. Soc. Ser B 50, pp. 157-224.

McCall, P.K. Richards and G.F. Walters (1977), Factors in software quality. Volumes 1, 2 and 3. Springfield Va., NTIS, AD/A-049-014/015/055.

Musa J. (1999). Software Reliability Engineering, McGraw Hill.

Neil M. (1999) IMPRESS (IMproving the software PRocESS using bayesian nets) EPSRC Project GR/L06683, http://www.csr.city.ac.uk/csr_city/projects/impress.html

Neil M., B. Littlewood and N. Fenton. (!996). "Applying Bayesian Belief Networks to Systems Dependability Assessment". Proceedings of Safety Critical Systems Club Symposium, Leeds, Published by Springer-Verlag.

Neil M., N. Fenton and L. Nielson (2000), "Building large-scale Bayesian Networks", Knowledge Engineering Review, 15(3), 257-284.

Pearl J. (1997) Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference Morgan Kauffman.

Towards the Verification and Validation of Online Learning Adaptive Systems

Ali Mili[1], Bojan Cukic[2], Yan Liu[2] and Rahma Ben Ayed[3]

[1]*College of Computing Science*
New Jersey Inst. of Technology, Newark, NJ 07102
mili@cis.njit.edu

[2]*Lane CSEE Department*
West Virginia University, Morgantown WV 26506-6109
{cukic,yanliu}@csee.wvu.edu

[3]*School of Engineering*
University of Tunis II, Belvedere 1002, Tunisia
rahma_k@yahoo.com

ABSTRACT

Online Adaptive Systems in general, and learning neural nets in particular cannot be validated using traditional verification and validation techniques, because they evolve over time, and past learning data influences their behavior. In this paper we discuss a framework for reasoning about online adaptive systems, and see how this framework can be used to perform V\&V on such systems.

KEYWORDS

Verification and Validation, Formal Methods, Refinement Calculi, On-Line Learning, Neural Networks, Adaptive Control.

1. INTRODUCTION: POSITION OF THE PROBLEM

1.1. On-Line Learning: An Emerging Paradigm

Adaptive Systems are systems whose function evolves over time, as they improve their performance through learning. The advantage of adaptive systems is that they can, through judicious learning, react to situations for which the designer did not make specific provisions. If learning and adaptation are allowed to occur after the control system if deployed, the system is called *online adaptive system.*

Online adaptive systems are attracting increasing attention in application domains where autonomy is an important feature, or where it is virtually impossible to analyze ahead of time all the possible combinations of environmental conditions that may arise. The controlled processes (as well as the control law) are often non-linear and subject to noise, disturbances, time delays and other un-modeled dynamics. Therefore, it is more advantageous to learn the system's behavior, rather than attempt its precise functional description. Examples of autonomous control applications are long term space missions where communication delays to ground stations are prohibitively long, and we have to depend on the systems' local capabilities to deal with unforeseen circumstances [15]. Examples of systems dealing with complex environmental conditions include flight control systems, which deal with a wide range of parameters, and a wide range of environmental factors. These systems must maintain flight safety and criticality equivalent to traditional human piloted systems. Other proposed applications include collision avoidance systems, multi-vehicle cooperative control, intelligent scheduling in manufacturing [11], control systems for automobile steering based on feature recognition in images [10], etc.

In recent years several experiments evaluated adaptive computational paradigms (neural networks, AI planners) for providing fault tolerance capabilities in control systems following sensor and/or actuator faults [28,29]. Experimental success suggests significant potential for future use. More recently, a family of neural networks, referred to as DCS (Dynamic Cell Structure) [16], have been used by NASA for on-line learning of aerodynamic derivatives [37] in a flight control system of an F-15. In the intelligent flight control system, the online neural learning DCS network provides the aircraft model's adaptation to the changes that may occur during the flight. The network is trained to the error in flight, i.e., the difference between the derivative values computed by a regression-based derivative estimator, and those provided by the preflight approximation algorithm (implemented by another neural network, which does not change in flight). The topology representing properties of the DCS network proved to be capable of providing the flight controller with the best available estimates of the aircraft's stability and control derivatives, while yielding a dramatically more compact way to store them. These advances were made possible by the fact that a DCS network eventually acquires ("learns") the connectivity structure, which represents the relation of topological proximity of points from the flight envelope.

The critical factor limiting wider use of neural networks and other soft-computing paradigms in process control applications, is our (in)ability to provide a theoretically sound and practical approach to their verification and validation. In this paper, we present a framework for reasoning about on-line learning systems, which we envision as a candidate technology for their verification and validation.

1.2. Verifying On-Line Learning Systems

While they hold great technological promise, on-line learning systems pose serious problems in terms of verification and validation, especially when viewed against the background of the tough verification standards that arise in their predominant application domains (flight control, mission control). Adaptive systems are inherently difficult to verify/ validate, *precisely because they are adaptive*. Specifically, consider that methods for software product verification are generally classified into three families [4]:

- *Fault Avoidance* methods, which are based on the premise that we can derive systems that are fault-free *by design*.
- *Fault Removal* methods, which concede that fault avoidance is unrealistic in practice, and are based on the premise that we can remove faults from systems after their design and implementation are complete.
- *Fault Tolerance* methods, which concede that neither fault avoidance nor fault removal are feasible in practice, and are based on the premise that we can take measures to ensure that residual faults do not cause failure.

Unfortunately, neither of these three methods is applicable *as-is* to adaptive systems, for the following reasons:

- *Fault Avoidance*. Formal design methods [14,20,27] are based on the premise that we can determine the functional properties of a system by the way we design it and implement it. While this holds for traditional systems, it does not hold for adaptive systems, since their design determines how they learn, but not what they will learn. In other words, the function computed by an online adaptive system depends not only on how the system is designed, but also on what data it has learned from.
- *Fault Removal: Verification*. Formal verification methods [1,25,22,26] are all based on the premise that we can infer functional properties of a software product from an analysis of its source text. While this holds for traditional systems, it does not hold for adaptive systems, whose behavior is also determined by their learning history.
- *Fault Removal: Testing*. All testing techniques [12,21,24] are based on the premise that the systems of interest will duplicate under field usage the behavior that they have exhibited under test. While this is true for traditional deterministic systems, it is untrue for adaptive systems, since the behavior of these systems evolves over time. We have observed in [2] that adaptive systems fail to meet this requirement (of maintaining or enhancing their behavior) even when they *converge*.
- *Fault Tolerance.* Fault tolerance techniques [3,32,33,36] are based on the premise that we have clear expectations about the functions of programs and programs parts, and use these expectations to design error detection and error recovery capabilities. With adaptive systems, it is not possible to formulate such expectations because the functions of programs/ program parts are not known at design time.

Because on-line learning systems are most often used in life-critical (e.g. flight control) and mission-critical (e.g. space) applications, they are subject to strict certification standards, leaving a wide technological gap between the requirements of the application domain and the capabilities of available technologies; our aim in this paper is to attempt to narrow this gap. First, we survey existing approaches.

1.3. Existing Approaches

Traditional literature typically describes adaptive computational paradigms with respect to their use, as function *approximators* or data classification tools. In most cases, their *correctness* is measured in terms of a misclassification rate on specific data sets, or by their ability to interpolate and/or extrapolate between known function values. This evaluation paradigm may work well only for applications where the system learns on a "training set" and remains unchanged in operational usage. In an attempt to discuss verification and validation of neural networks, LiMin Fu [17] interprets verification to refer to correctness and interprets validation to refer to accuracy and efficiency. He establishes correctness by analyzing the process of designing the neural network, rather than the functional properties of the final product. An intuitively similar, but more elaborate approach has been described by Gerald Peterson [31]. Peterson describes the opportunities for verification and validation of neural networks in terms of the activities in their development life-cycle, as shown in Figure 1.

If a problem is judged to be solvable by a neural network (feasibility phase), training data is gathered. Verification of the training data includes the analysis of appropriateness and comprehensiveness. This step is not fully applicable to on-line learning applications since training data are related to the real-time evolution of the system state, rather than the design choice. Verification of the training process typically examines the convergence properties of the learning algorithm in terms of achieving the desired optimal problem solution. Evaluation of interpolation and extrapolation capabilities of the network and domain specific verification activities set the stage for the overall verification and validation. The strong emphasis on domain specific knowledge, its formal representation and mathematical analysis is suggested in [19] too. Del Gobbo and Cukic propose the analysis of the neural network with respect to conditions implying the existence of the solution (for function approximation) and the reachability of the solution from any possible initial state. Their third condition can be interpreted as condition for preservation of the learned information.

While meaningful and well organized, Peterson's approach provides little guidance on the choice of specific rigorous V&V techniques. Proposed techniques are mostly based on empirical evaluation through simulation and/or experimental testing. In an on-going effort, a group of researchers at

NASA Ames Research Center are defining life-cycle V&V methods applicable to systems which have (an) integrated adaptive software component(s) [7]. In some cases, neural networks are modified to provide support for testing based (or on-line) validation of results. For example, Leonard et. al. [23] suggest a new architecture called *Validity Index*. A *Validity Index* network is a derivative of Radial Basis Function (RBF) network with the additional ability to calculate confidence intervals for its predictions based on the probability density of the "similar" training data observed in the past.

In a recent survey of methods for validating on-line learning neural networks, O. Raz [34] calls this approach *on-line monitoring and novelty detection* and attributes to it a significant potential for the future use. The other promising research direction, according to Raz, is periodic rule extraction from an on-line neural network and partial (incremental) re-verification of these rules using symbolic model checking. Practical hurdles associated with this approach include determining the frequency of rule extraction and impracticality of near real-time model checking of complex systems. LiMin Fu [17] discuss the verification and validation of neural nets, where he interprets verification to refer to correctness and interprets validation to refer to accuracy and efficiency. He establishes correctness by analyzing the process of designing the neural net, rather than the functional properties of the final product.

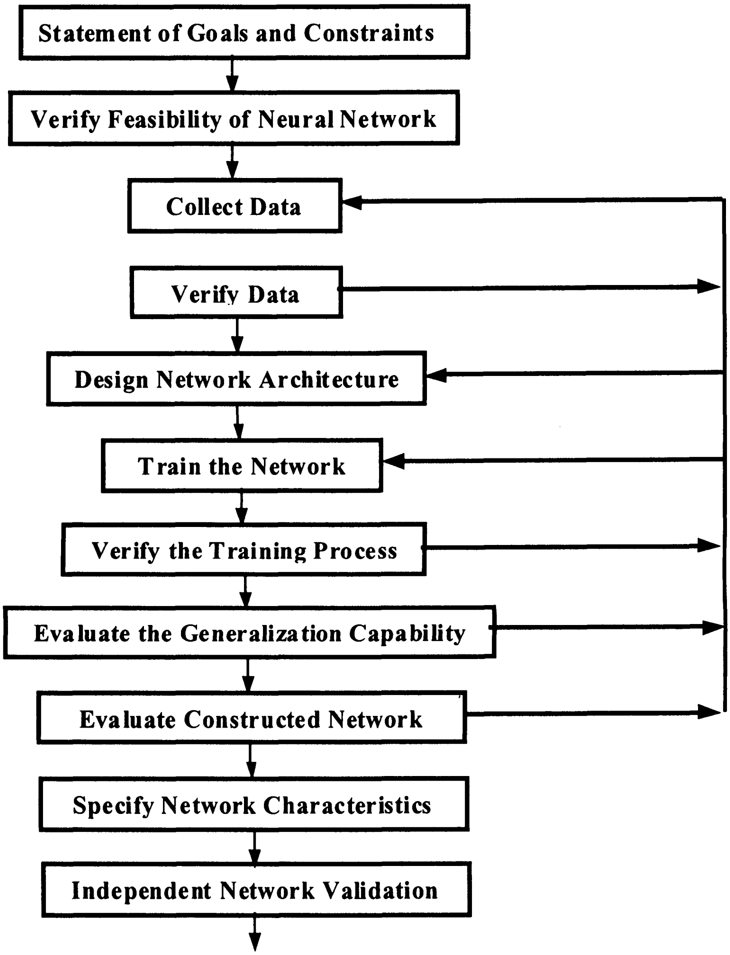

Statement of Goals and Constraints
Verify Feasibility of Neural Network
Collect Data
Verify Data
Design Network Architecture
Train the Network
Verify the Training Process
Evaluate the Generalization Capability
Evaluate Constructed Network
Specify Network Characteristics
Independent Network Validation

Figure 1. Neural Network Construction Lifecycle.

2. TENETS OF A REFINEMENT-BASED APPROACH

2.1. Characterizing Our Approach

Our approach to the verification of on-line learning systems can be summarized in the following premises:

- We establish the correctness of the system, not by analyzing the process by which the system has been designed, but rather by analyzing the functional properties of the final product, and the evolution of these systems can be controlled to preserve/ enhance selected properties.
- Qualifying the first premise, we capture the functional properties of the system not by the exact function that the system defines at any stage in

its learning process, but rather by a *functional envelope*, which captures the range of possible functions of the system for a given learning history. This concept will be more formally defined in section 3.1.

- In order to make testing meaningful, we need to ensure that the system evolves in a way that preserves or enhance its behavior under test. We call this *monotonic learning*, and we investigate it briefly in section 4.1. Of course, on-line learning systems are supposed to get better as they acquire more learning data, but our definition of better is very specific: it means that the functional envelope of the system grows increasingly more refined with learning data (in the sense of refinement calculi [5,8,13,18,39]).
- In order to support some form of correctness verification, we must recognize that the variability of learning data and the focus on functional envelope (rather than precise function) weaken considerably the kinds of functional properties that can be established by correctness verification. Typically, all we can prove are minimal safety conditions; we refer to this as *safe learning* (proving that learning preserves safety conditions), and we discuss it briefly in section 4.2.

In the sequel, we introduce some mathematical background, which we use in the remainder of the paper.

2.2. Specification Structures

The verification and validation of systems, whether adaptive or not, can only be carried out with respect to predefined functional properties, which we capture in *specifications*. In this paper, we model specifications by means of binary relations. A *relation* R from set X to set Y is a subset of the Cartesian product $X \times Y$. A *homogeneous* relation on S is a relation from S to S. We use relations to represent specifications. Among relational constants we cite the *identity relation*, denoted by I, and the *universal relation*, denoted by L. Among operations on relations we cite the *product*, which we represent by $R \circ R'$ or by RR' (when no ambiguity arises), the *complement*, which we represent by $\overline{R}$, the *inverse*, which we represent by $\widehat{R}$, and the set theoretic operations of *union* and *intersection*.

We wish to introduce an ordering between (relational) specifications to the effect that a specification is greater than another specification if and only if it captures stronger functional requirements. We refer to this ordering as the *refinement ordering*, we denote it by $R \sqsupseteq R'$ and we define it as

$$RL \, \mathrm{I} \, R'L \, \mathrm{I} \, (R \, \mathrm{Y} \, R') = R'.$$

The following definition and proposition give the reader some intuition for the meaning of the refinement ordering.

Definition 1

A program P on space S is said to be correct with respect to specification R on S if and only if $\boxed{P} \sqsupseteq R$ where $\boxed{P}$ is the function defined by program P.

Proposition 1

Specification R refines specification R' if and only if any program correct with respect to R is correct with respect to R'.

Figure 2 illustrates, in set theoretic terms, the meaning of the refinement ordering, by showing the graphs of two relations R and R' on the same sets. R refines R' because it has a larger domain and has fewer images for each argument. By contrast, Q does not refine Q' nor does Q' refine Q.

As a complement to studying ordering properties of the refinement relation, we also investigate lattice properties [9]. In [6], we have derived two propositions pertaining to the lattice properties of the refinement ordering. We present them here without proof, but with some discussion of their intuitive meaning.

Proposition 2

Two relations R and R' have a least upper bound (also called the join) with respect to the refinement ordering if and only if they satisfy the condition (called the consistent condition):

$$RL\,\mathrm{I}\ R'L = (R\,\mathrm{I}\ R')L.$$

When they do satisfy this condition, their join is denoted by $(R \sqcup R')$ and is defined by

$$(R \sqcup R') = R\,\mathrm{I}\ \overline{R'L}\,\mathrm{Y}\,R'\,\mathrm{I}\ \overline{RL}\,\mathrm{Y}(R\,\mathrm{I}\ R').$$

The consistency condition means that R and R' can be satisfied (refined) simultaneously, i.e. that they have an upper bound. As for the expression of the join, suffice it to say that $(R \sqcup R')$ represents the specification that captures all the functional features of R (upper bound of R) and all the functional features of R' (upper bound of R') and nothing more (*least* upper bound). A crucial property of joins, for our purposes, is that an element A refines $R \sqcup R'$ if and only if it refines simultaneously R and R . In other words, the join of R and represents the *sum* of all the functional features of R and R'. This sum can be derived only if R and R' do not contradict each other (re: the consistency condition). We argue in [6] that complex specifications can be structured in terms of simpler sub-specifications using the join operator.

In addition to discussing least upper bounds (joins), we also discuss greatest lower bounds (meets), which are introduced in the following proposition.

Proposition 3

Any two relations R and R' have a greatest lower bound (also called the meet), which is denoted by $(R \sqcap R')$ and defined by

$$(R \sqcap R') = RL\,\mathrm{I}\;\; R'L\,\mathrm{I}\;\; (R\,\mathrm{Y}\,R').$$

The meet of two relations R and R' is a specification that refined by R (lower bound of R), refined by R' (lower bound of R'), and is maximal (greatest lower bound): in other words, it captures all the functional features that are common to R and R'.

The following lemma, which presents trivial lattice identities, will be generalized later for our purposes.

Lemma 1

The following identities hold in any lattice:

- $(A \sqsupseteq B) \vee (A \sqsupseteq C)$ *logically implies* $A \sqsupseteq (B \sqcap C)$
- $(B \sqsupseteq A) \vee (C \sqsupseteq A)$ *logically implies* $(B \sqcap C) \sqsupseteq A$

The first clause stems readily from the transitivity of the refinement ordering, and the lattice identities. The second clause can be proved by observing that the left hand side provides that is a A lower bound for B and C, hence it is refined by the greatest lower bound.

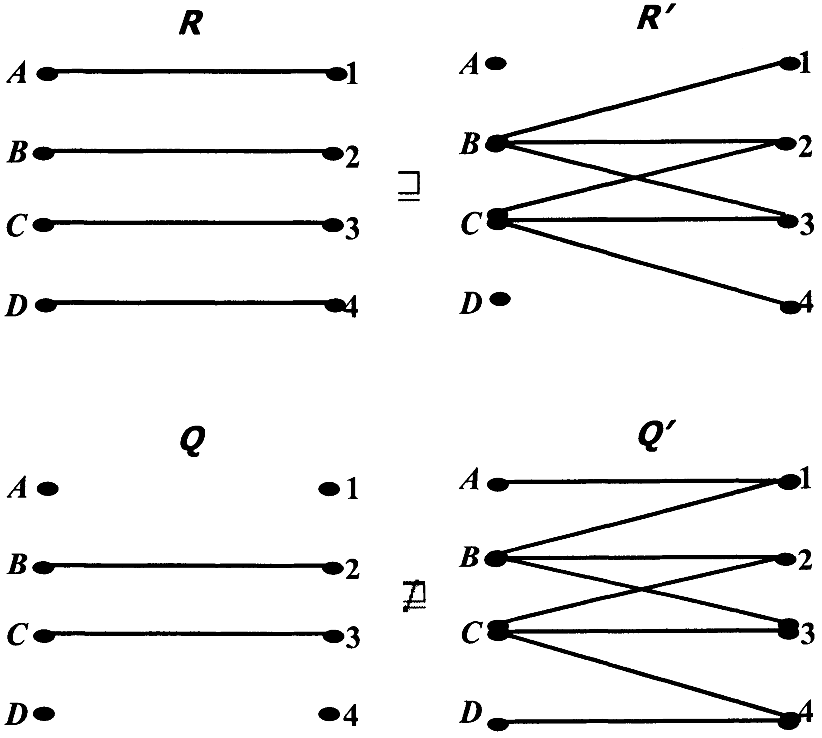


Figure 2. Refinement Ordering in Pictures.

3. A COMPUTATIONAL MODEL FOR ON-LINE LEARNING SYSTEMS

3.1. An Abstract Model

Before we discuss the specifics of the verification methods we propose, we first introduce an abstract computational model for adaptive systems and their evolution through learning. Figure 3 depicts the abstract model we have of an online adaptive system; this model is purposefully generic, to support a wide range of possible implementations (RBF, DCS, MLP), and to enable us to focus on relevant computational features (as opposedto being distracted by implementation specific details). Our model includes the following features:

- Set X represents the set of inputs that may be submitted to the adaptive system.
- Set Y represents the set of outputs that the adaptive system may return as output.

- Set H represents the set of learning data histories that are submitted to the adaptive system for learning; typically, this set is nothing but the set of sequences of the form (x,y), where $x \in X$ and $y \in Y$. We let ε represent the empty sequence (as an element of H).
- Function F is the function which, to each learning history h in H associates a function F_h from X to Y that captures the behavior of the adaptive system after receiving learning data h. According to this definition, the initial behavior of the adaptive system before any learning history is received is F_ε.
- Function R is the function which, to each learning history h in H associates a relation R_h from X to Y that captures the *learned behavior* of data h, and nothing else. Whereas F_h may include behavior that stems from its initialization, or stems from extrapolations, or stems from default options, R_h remains undefined or under-defined until learning data intervenes.

In order to elucidate the meaning of relation R_h, for history h, we consider the following development scenario for adaptive systems. An adaptive system is defined by some learning rule, which maps a learning history h into a function F_h; the learning algorithm is also defined by means of implementation-specific parameters, including randomly chosen parameters. For the sake of abstraction, we denote the vector of implementation-specific parameters by a variable, say λ, and we let Λ be the set of possible values for λ. To fix our ideas, we can think of Λ as representing a family of possible implementations of the learning algorithm, and of λ as a specific implementation within the selected family; also, we denote by F_h^λ the function that captures the behavior of the adaptive system whose parameters vector is λ, upon receiving learning data h. With this background in mind, we let R_h be defined as follows:

$$R_h = \sqcap_{\lambda \in \Lambda} F_h^\lambda$$

By virtue of the definition of meet, $\sqcap_{\lambda \in \Lambda} F_h^\lambda$ can be interpreted to represent the functional information that is common to all possible implementations of the learning algorithm, for all possible values of λ. While F_h^λ is dependent on λ, R_h is dependent on Λ.

As a corollary of this definition, consider the initial values of F_h^λ and R_h for $h = \varepsilon$, i.e. at the beginning of the learning process. Whereas F_ε^λ represents the (mostly arbitrary) initialization of the function of the adaptive system, R_ε represents the information that all instances of F_ε^λ, for all values of λ in Λ have in common. In effect, R_ε captures all the functional information that stems from Λ, and that is specific to the family of learning algorithms being used.

The definition of R_h yields the following proposition, which we present without proof (the proof is a trivial lattice identity).

Proposition 4

For all $\lambda \in \Lambda$, we have:

$$\forall h : F_h^\lambda \sqsupseteq R_h .$$

This proposition stems readily from the definition of R_h as the meet of all F_h, for all h: the meet of many terms is lower than any one term.

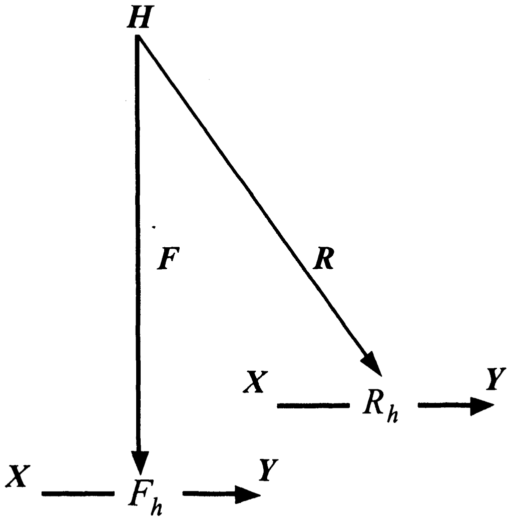


Figure 3. Abstract Computational Model.

3.2. A Concrete Model: The Back-Propagation Learning Algorithm

In this section, we consider the back-propagation learning algorithm, and we analyze it to show that it fits the abstract computational model that we have presented above. The back-propagation algorithm was first developed by Werbos in 1974 [38] but attracted little attention initially. It was later independently rediscovered by Parker [30] in 1982 and by Rumelhart, Hinton and Williams [35] in 1986. The version we present below, taken from [17], is due to [35].

- Weight Initialization. Set all weights and node thresholds to small random numbers. Note that the node threshold is the negative of the weight from the bias unit (whose activation level is fixed at 1).
- Calculation of Activation.
  1. The activation level of an input unit is determined by the instance presented to the network.
  2. The activation level O_j of a hidden and output unit is determined by

     $$O_j = \sigma(\sum W_{ji}O_i - \theta_j),$$

     where W_{ji} is the weight from an input O_i, θ_j is the node threshold, and σ is the sigmoid function:

     $$\sigma(\alpha) = \frac{1}{1+e^{-\alpha}}$$

- Weight Training.
  1. Start at the output units and work backward to the hidden layers recursively. Adjust weights by

     $$W_{ji}(t+1) = W_{ji}(t) + \Delta W_{ji},$$

     where $W_{ji}(t)$ is the weight from unit i to unit j at time t and ΔW_{ji} is the weight adjustment.
  3. The weight change is computed by

     $$\Delta W_{ji} = \eta\delta_j O_i,$$

     where η is a trial-independent learning rate $0 < \theta < 1$ and δ_j is the error gradient at unit j. Convergence is sometimes faster by adding a momentum term:

     $$W_{ji}(t+1) = W_{ji}(t) + \eta\delta_j O_i + \alpha(W_{ji}(t) - W_{ji}(t-1)),$$

where $0 < \alpha < 1$.

4. The error gradient is given by:
   - For the output units:

     $$\delta_j = O_j(1 - O_j)(T_j - O_j),$$

     where T_j is the desired (target) output activation and O_j is the actual output activation at output unit j.
   - For the hidden units:

     $$\delta_j = O_j(1 - O_j)\sum_k \delta_k W_{kj},$$

     where δ_k is the error gradient at unit k to which a connection points from hidden unit j.
5. Repeat iterations until convergence in terms of the selected error criterion. An iteration includes presenting an instance, calculating activations, and modifying weights.

We interpret this algorithm as defining function F_h (see section 3) by induction on the complexity (length) of h. If we recognize that F_h is not entirely determined by h but is also dependent on the arbitrary initial parameters (and their subsequent manipulations) then we rewrite this function as F_h^λ, where λ is the vector of weights

$$\lambda = \begin{pmatrix} \cdots \\ \cdots \\ W_{ji} \\ \cdots \\ \cdots \end{pmatrix}$$

Also, we recognize that the range of values that weights can take evolves as the algorithm proceeds, hence the term Λ in the equations of section 3 should, in fact, be indexed with h; to acknowledge this, we write it as Λ_h. Consequently, we find:

- Λ_ε, the initial set of possible weights, is defined by the *Weight Initialization* step in the back-propagation algorithm. The initial values of the weights are usually chosen rather small, since large weights cause the activation functions to saturate early, and cause the network to be stuck in a very flat plateau or a local minimum near the starting point. Typically, the initial values of weights are chosen as random values uniformly distributed between $\frac{-0.5}{FanIn}$ and $\frac{+0.5}{FanIn}$, where $FanIn$ of a unit is the number of units which are fed forward into this unit [17].

- $\Lambda_{h\bullet(x,y)}$ is obtained from Λ_h by applying the function detailed in the *Weight Training* step of the back-propagation algorithm. Specifically, if we let WT be the function detailed in this step, which has the form

$$\begin{pmatrix} W_{ji}(t+1) \\ \cdots \\ \delta_j \end{pmatrix} = WT \begin{pmatrix} W_{ji}(t) \\ \cdots \\ \delta_j \\ O_j \end{pmatrix}.$$

then $\Lambda_{h\bullet(x,y)}$ can be defined as follows:

$$\Lambda_{h\bullet(x,y)} = \{ \begin{pmatrix} \cdots \\ W_{ji}(t+1) \\ \cdots \end{pmatrix} \mid \begin{pmatrix} W_{ji}(t+1) \\ \cdots \\ \delta_j \end{pmatrix} = WT \begin{pmatrix} W_{ji}(t) \\ \cdots \\ \delta_j \\ O_j \end{pmatrix} \wedge \begin{pmatrix} \cdots \\ W_{ji}(t) \\ \cdots \end{pmatrix} \in \Lambda_h \}$$

In light of this, we rewrite the characterization of R_h as follows:

$$R_h = \sqcap_{\lambda \in \Lambda_h} F_h^{\lambda}$$

In particular, if we take $h = \varepsilon$, we find that Λ_ε is the set of all admissible initial weights, and R_ε is the meet of all possible functions F_ε^{λ} for all admissible initial weights. Under some weak conditions (which are discussed in the sequel), we find a simple expression for R_ε:

$$R_\varepsilon = \{(x,y) \mid \exists \lambda : (x,y) \in F_\varepsilon^{\lambda}\}.$$

This formula is intuitively appealing: R_ε is the set of all input output pairs (x,y) such that (x,y) is in F_ε^{λ} for some admissible initial weighting λ. Note that while F_ε^{λ} reflects the arbitrary choice of an initial weighting, R_ε does not; it only reflects the learning algorithm and the specific network architecture. More generally, we intend R_h to reflect the learning algorithm, the network architecture, and the learning data – *but not to reflect any arbitrary choice of random weights*. Note also, on the expression above, that while F_h^λ is deterministic, R_ε is (very) non-deterministic; R_h is obtained from F_h^λ by abstracting away the arbitrary determinacy of F_h^λ.

In order to assess the variability of the system function with respect to the choice of initial weights, we have run an experiment on a simple back-propagation neural network with one hidden layer, and have submitted to it learning data about the *exclusive or* function. Also, we have selected the initial weights, and have observed how these affect the function F_h^λ for various values of h. Specifically, h is a sequence of *epochs*, where each epoch is made up of the four sets of inputs (combinations of two Boolean variables) along with their corresponding outputs by the *exclusive or* function. The column labeled "10" in figure 4 represents the learning sequence h made up of ten epochs. By abuse of notation, we can represent h by the number of epochs in h. We can make the following observations:

- The initial weights have a large impact on the evolution of F_h^λ.
- This impact lasts well into the future, and does not completely disappear even after several thousand epochs.

For the purposes of our study, this means that R_h remains distinct from F_h^λ even for a long learning sequence h. Figure 4 also allows us to visualize the difference between F_h^λ and R_h: For example F_h^λ maps input (1,1) into 0.95718, whereas R_h also maps it into, among others,

0.70029, 0.50565, 0.51080.

	(1,1)	0.95718	0.88763	0.64715	0.57213	0.50737	0.12861	0.04999388
	(1,0)	0.88929	0.75727	0.49946	0.48526	0.51578	0.88881	0.95669980
	(0,1)	0.88985	0.76131	0.50934	0.48949	0.51579	0.88868	0.95675480
	(0,0)	0.74329	0.58170	0.41602	0.45580	0.50102	0.09960	0.03909299
W_0=0.5								7560**
	(1,1)	0.70029	0.58756	0.53496	0.52377	0.51087	0.14863	0.04999296
	(1,0)	0.60074	0.51103	0.48290	0.48596	0.49385	0.87160	0.95670090
	(0,1)	0.60987	0.52137	0.48944	0.48869	0.49424	0.87141	0.95675580
	(0,0)	0.55051	0.49504	0.48686	0.49964	0.51760	0.11487	0.03909090
W_0=0.0								8926**
	(1,1)	0.50565	0.50740	0.50880	0.50976	0.51116	0.50880	0.04999402
	(1,0)	0.48353	0.48525	0.48714	0.48836	0.48878	0.49985	0.95669870
	(0,1)	0.49364	0.49367	0.49203	0.49037	0.48888	0.50006	0.95675415
	(0,0)	0.51421	0.51434	0.51342	0.51227	0.51134	0.51613	0.03909125
W_0 *								8942**
	(1,1)	0.51080	0.51098	0.51101	0.51096	0.51118	0.50909	0.04999226
	(1,0)	0.48304	0.48416	0.48627	0.48794	0.48876	0.49888	0.95670134
	(0,1)	0.49480	0.49397	0.49197	0.49027	0.48885	0.49911	0.95675653
	(0,0)	0.51010	0.51027	0.51051	0.51073	0.51137	0.51662	0.03908928

* : Random values range from -0.3 to +0.3.

** : Iteration times when network comes to convergence.

Figure 4. One Hidden Layer MLP NN for XOR Problem Trained by BP Algorithm with Different Initial Weights.

4. VERIFICATION OF ON-LINE LEARNING SYSTEMS

Given that we have derived the *functional envelope* of a an on-line learning system (as relation R_h), we discuss now how we can infer functional properties of the system. We discuss two methods in turn: *Monotonic Learning* and *Safe Learning*.

4.1. Monotonic Learning

The idea of *monotonic learning* is to ensure that the adaptive system learns in a monotonic fashion, so that whatever claims we can make about the behavior of the system prior to its deployment are upheld while the system evolves through learning. Of course, we can hardly expect F_h to be monotonic with respect to h, since there is no way to discriminate between information of F_h that stems from learning and information that stems from arbitrary choices. In addition, whenever F_h^λ is total (which is fairly common), it is in fact maximal in the refinement ordering, hence cannot be further refined. We can, however, expect R_h to be monotonic, in the following sense.

Definition 2

An adaptive system is said to exhibit monotonic learning if and only if for all h in H, and for all (x,y) in $X \times Y$,

$$R_{h.(x,y)} \sqsupseteq R_h$$

where $h \cdot (x,y)$ is the sequence obtained by concatenating h with (x,y).

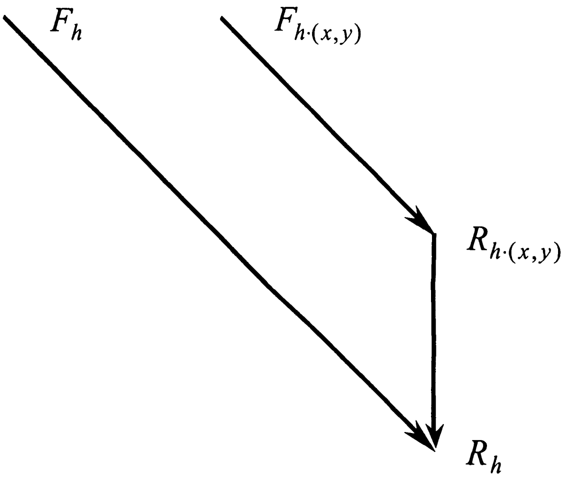


Figure 5. Monotonic Learning Increases R_h, Not Necessarily F_h.

Figure 5 illustrates in what sense monotonicity of R does not necessarily imply monotonicity of F. The challenge of this approach is to analyze what kinds of restrictions we must impose on the learning algorithm in order to ensure the monotonicity of R, or, alternatively, what kinds of learning algorithms ensure this property. Note that the refinement ordering is reflexive, hence nothing precludes us from a situation where $R_{h.(x,y)} = R_h$. One possible way to ensure monotonicity is to compare $R_{h.(x,y)}$ against R_h for refinement, and to discard (x,y) whenever the former does not strictly refine the latter. In practice this will work only if discarding learning data is an exceptional occurrence, rather than a routine occurrence.

The interest of monotonic learning is that whatever properties can be established by analyzing the adaptive system at any stage of its learning are sure to be preserved (in the sense of refinement) as the system learns. In particular, all the properties of R_ε (before learning starts) are maintained as the system learns. More significantly, any behavior that is exhibited at the testing phase is sure to be preserved (i.e. duplicated or refined) in field usage.

Traditional certification algorithms observe the behavior of a software product under test, and make probabilistic/ statistical inferences on the operational attributes of the product (reliability, availability, etc). The crucial hypothesis on which these probabilistic/ statistical arguments are based is that the software product will reproduce under field usage the behavior that it has exhibited under test. This hypothesis does not hold for adaptive neural nets, because they evolve their behavior (learn) as they practice their function. Of course, one may argue that they evolve their behavior for the better; but *better* in the sense of a neural net (convergence, stability) is not necessarily better in the sense of correctness verification (monotonicity with respect to the refinement ordering). Concretely, a neural net may very well satisfy the test oracle in the testing phase, and fail to satisfy it in the field usage phase, even though it converges. See Figure 6.

In principle, to apply monotonic learning we need to derive a closed form expression of R_h, then we derive the condition provided in definition 2 and prove it. Because it is rather impractical to derive a closed form of R_h, this approach is unrealistic. As a substitute, we submit sufficient conditions for monotonic learning, starting with the following proposition.

Proposition 5

If the following conditions holds,

$$\forall \lambda \exists \lambda' : F^{\lambda}_{h.(x,y)} \sqsupseteq F^{\lambda'}_{h},$$

then the pair (x, y) *provides monotonic learning with respect to learning history* h.

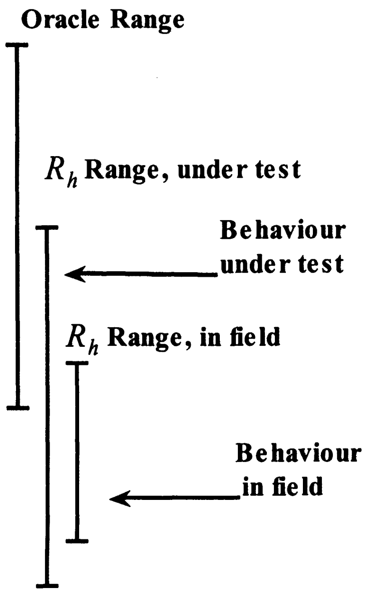


Figure 6. Convergence does Not Ensure Monotonicity.

Proof. We must prove that under the condition cited above,

$$R_{h.(x,y)} \sqsupseteq R_h$$

To this effect, we proceed by logical implications, starting from our hypothesis.

$$\forall\lambda\exists\lambda': F^{\lambda}_{h.(x,y)} \sqsupseteq F^{\lambda'}_{h}$$

$$\Rightarrow \{\text{ Definition of meet, transitivity}\}$$

$$\forall\lambda\exists\lambda': F^{\lambda}_{h.(x,y)} \sqsupseteq \sqcap\ \lambda' \in \Lambda F_{\lambda'}$$

$$\Rightarrow \{\text{ Definition }\}$$

$$\forall\lambda\exists\lambda': F^{\lambda}_{h.(x,y)} \sqsupseteq R_h$$

$$\Rightarrow \{\text{ Lattice identity }\}$$

$$\forall\lambda\exists\lambda': \sqcap\ \lambda F^{\lambda}_{h\cdot(x,y)} \sqsupseteq R_h$$

$$\Rightarrow \{\text{ Definition }\}$$

$$R_{h.(x,y)} \sqsupseteq R_h.$$

qed

We have found that often, function F_h^{λ} is total for all h and all λ ; this gives weight to the following proposition, which gives another (weaker, but no less general) sufficient condition of monotonic learning.

Proposition 6

If F_h^{λ} is total for any history h and any initial weights λ, and the following condition holds,

$$\forall \lambda \exists \lambda' : F^{\lambda}_{h.(x,y)} \sqsupseteq F^{\lambda'}_{h},$$

then the pair (x, y) provides monotonic learning with respect to learning history h.

Proof. We consider the condition

$$\forall \lambda \exists \lambda' : F^{\lambda}_{h.(x,y)} \sqsupseteq F^{\lambda'}_{h},$$

Because both terms of this inequation are function, this condition is equivalent to

$$\forall \lambda \exists \lambda' : F^{\lambda}_{h.(x,y)} \supseteq F^{\lambda'}_{h},$$

Because both functions are total, this condition can further be simplified as:

$$\forall \lambda \exists \lambda' : F^{\lambda}_{h.(x,y)} = F^{\lambda'}_{h}$$

qed

In other words, the learning pair (x, y) produces monotonic learning if and only if appending to learning history h produces the same outcome as starting with some other initial weight λ' and applying the learning history h. Presumably, λ' would have been a better initial weight than λ, since we get the same function for one less learning pair. This condition is not suggesting to choose a better λ, but rather is giving a sense to our concept of monotonic learning, which provides that as we learn more and more (i.e. as h increases in length), the range of possible values for function F_h^{λ} decreases. Note that there is no condition to the effect that every value of F_h^{λ} can be attained (by means of changing λ) for history $h \cdot (x, y)$; hence the condition of corollary 5 is ensuring that the range of possible values for

F_h^λ (which is the range of relation R_h) shrinks as h expands. We will discuss applications of this proposition in section 5.

4.2. Safe Learning

The main idea of *safe learning* is to ensure that as the adaptive system evolves through learning, it maintains some minimal safety property S. In other words, in addition to maintaining the identity

$$\forall h \forall \lambda, F_h^\lambda \sqsupseteq R_h,$$

which stems from the modeling of the system, we also require that the system maintains the following property

$$\forall h, F_h \sqsupseteq S$$

to ensure the safe operation of the adaptive system as it evolves through learning. By virtue of the lattice-like structure of the refinement ordering, we infer that F must satisfy:

$$\forall h \forall \lambda, F_h^\lambda \sqsupseteq (R_h \sqcup ').$$

See figure 7.

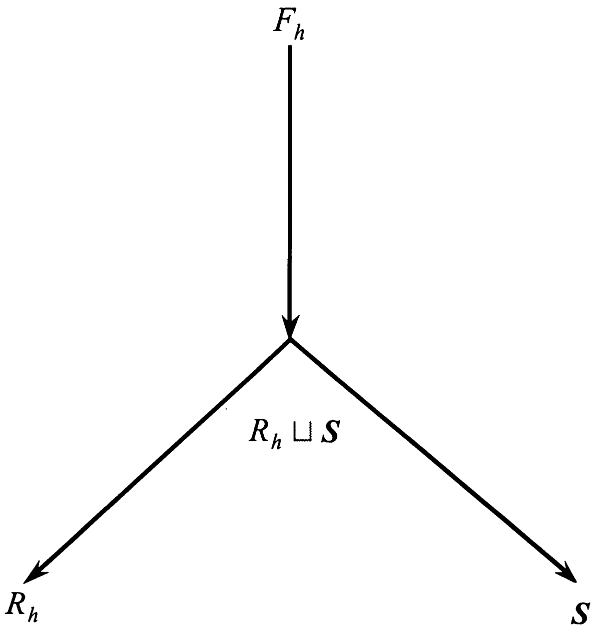


Figure 7. Safe Learning.

This can be satisfied if and only if R_h and S do indeed have a join, i.e. if and only if they satisfy the consistency condition. The aggregate of conditions that characterize the safe learning of the adaptive system can be written as:

$$\forall h, R_h L \mathrm{I}\ \ SL = (R_h \mathrm{I}\ \ S)L$$

$$\forall h, F_h \sqsupseteq (R_h \sqcup S).$$

These conditions can be maintained by placing restrictions on the learning algorithms that can be deployed, or by controlling learning data that gets fed into the adaptive system, as per the following inductive argument:

2. As the basis of induction, these conditions hold for $h = \varepsilon$, since is the minimal element of the lattice of refinement.

6. Given that they hold for h, we can ensure that they hold for $h.(x, y)$ by accepting entry (x, y) only if it does not violate these conditions.

4.3. Inductive Alternatives

Most traditional program verification methods tackle the complexity of the task at hand by doing induction on some dimension of program structure (control structure, data structure, depth of recursion, etc). Likewise, while the two methods we present here appear attractive, we have no doubt that they are complex in practice, because they rely on an explicit formulation of the functional envelope of the system. Hence we are focusing our attention on means to use induction in such a way that we can apply these methods without having to derive R_h. The key to the inductive approach is the ability to derive inductive relationships between R_h and $R_{h.(x,y)}$. In the case of the neural net we have discussed in section 3.2, we know the relation between successive weights (as defined by the *Weight Training* function, WT), the relation between a set of weights (λ) and the corresponding system function (F_h^λ) and we know how the functional envelope R_h is derived from system functions (by taking the meet for all values of λ). We must infer from this the relation between R_h and $R_{h.(x,y)}$. See figure 8.

$$\begin{tikzcd}
\lambda_0, \varepsilon \arrow[r, "F"] \arrow[d, "WT^*"] & F_{\varepsilon}^{\lambda_0} \arrow[r, "\sqcap"] & R_h \\
\lambda, h \arrow[r, "F"] \arrow[d, "WT"] & F_h^{\lambda} \arrow[r, "\sqcap"] & R_h \arrow[d, "?"] \\
WT(\lambda), h \cdot (x, y) \arrow[r, "F"] & F_{h\cdot(x,y)}^{WT(\lambda)} \arrow[r, "\sqcap"] & R_{h\cdot(x,y)}
\end{tikzcd}$$

Figure 8. Inductive Structure.

5. ILLUSTRATION: A SIMPLE MULTI LAYER PERCEPTRON

We consider a simple Multi Layer Perceptron (MLP) with the simplest of architectures: one input layer, one hidden layer and one output layer, each containing a single neuron; see Figure 9. We want to use this example to discuss the condition of monotonicity; to this effect, we first write the expression of F_h^{λ}. We find,

$$F_h^{\lambda} = \{(x, y) \mid y = \sigma(w_2 \times \sigma(w_1 \times x))\}$$

where

- Function σ is defined by

$$\sigma(t) = \frac{1}{1+e^{-t}}.$$

- The vector $\begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$ obtained by backpropagation starting from initial weights λ, after the learning sequence h.

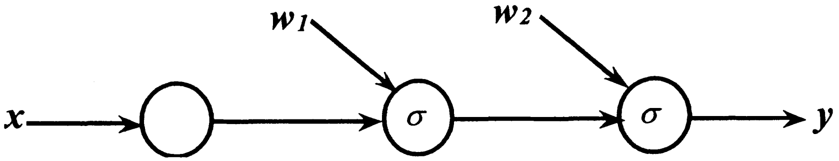


Figure 9. Architecture of a (Very) Simple Multi Layer Perceptron.

In order to articulate how the vector of weights $\begin{pmatrix} w_1 \\ w_2 \end{pmatrix}$ is derived from the initial weights (λ) and from the learning history (h), we write

$$\begin{pmatrix} w_1 \\ w_2 \end{pmatrix} = W_h(\lambda)$$

where function W_h is defined inductively (on h) by

- $W_\varepsilon(\lambda) = \lambda$.
- $W_{h\cdot(x,y)}(\lambda) = WT(W_h(\lambda), \begin{pmatrix} x \\ y \end{pmatrix})$,

where WT is, in turn, defined by

$$WT(\begin{pmatrix} w_1 \\ w_2 \end{pmatrix}, \begin{pmatrix} x \\ y \end{pmatrix}) =$$

$$\begin{pmatrix} w_1 + \eta x\sigma(w_1x)(1-\sigma(w_1x))\sigma(w_2\sigma(w_1x))(1-\sigma(w_2\sigma(w_1x)))(y-\sigma(w_2\sigma(w_1x)))w_2 \\ w_2 + \eta\sigma(w_1x)\sigma(w_2\sigma(w_1x))(1-\sigma(w_2\sigma(w_1x)))(y-\sigma(w_2\sigma(w_1x))) \end{pmatrix}$$

where η is the learning rate.

By inspection of the formula of F_h^λ, we infer that F_h^λ is total (since σ is total), hence we use proposition 6, which provides the following sufficient condition for monotonicity:

$$\forall\lambda\exists\lambda': F_{h.(x,y)}^{\lambda} = F_h^{\lambda'}.$$

We interpret this condition as:

$$\forall\lambda\exists\lambda': (\forall t: F_{h.(x,y)}^{\lambda}(t) = F_h^{\lambda'}(t)).$$

Referring back to the formula of F_h^λ, we find that a sufficient (perhaps also necessary) condition of monotonicity is:

$$\forall\lambda\exists\lambda': W_{h.(x,y)}(\lambda) = W_h(\lambda').$$

In the sequel, we characterize cases under which this condition is satisfied; for each case, we present a brief argument, then discuss the significance of the case.

- *The first learning pair produces monotonicity.* If we take $h=\varepsilon$, we find:

  $$\forall\lambda\exists\lambda': W_{h\cdot(x,y)}(\lambda) = W_h(\lambda')$$

  $\Leftrightarrow$ { Because $h=\varepsilon$ }

  $$\forall\lambda\exists\lambda': W_{h\cdot(x,y)}(\lambda) = \lambda'$$

  $\Leftrightarrow$ { By definition of W_h }

  $$\forall\lambda\exists\lambda': WT(W_h(\lambda),\begin{pmatrix}x\\y\end{pmatrix}) = \lambda'$$

  $\Leftrightarrow$ { Because $h=\varepsilon$ }

  $$\forall\lambda\exists\lambda': WT(\lambda,\begin{pmatrix}x\\y\end{pmatrix}) = \lambda'$$

  $\Leftrightarrow$ { WT is a total function}

  `true`

  Given that monotonicity means in effect that the new learning pair refines (enhances) prior knowledge, there is no doubt that the first pair always does (by contrast with subsequent pairs, which may conflict with prior knowledge).

- *Duplication produces monotonicity.* If we let h be a sequence of length 1, and let the new learning pair be a copy of the first pair, then we satisfy the condition of monotonicity. Formally,

  $$\forall\lambda\exists\lambda': W_{h\cdot(x,y)}(\lambda) = W_h(\lambda')$$

  $\Leftrightarrow$ { By definition of W_h }

  $$\forall\lambda\exists\lambda': WT(W_h(\lambda),\begin{pmatrix}x\\y\end{pmatrix}) = W_h(\lambda')$$

  $\Leftrightarrow$ { Because $h=(x,y)=\varepsilon\cdot(x,y)$ }

$$\forall\lambda\exists\lambda': WT(W_h(\lambda),\begin{pmatrix}x\\y\end{pmatrix}) = WT(W_\varepsilon(\lambda'),\begin{pmatrix}x\\y\end{pmatrix})$$

$$\Leftrightarrow \{ \text{By definition of } W_h \ \}$$

$$\forall\lambda\exists\lambda': WT(W_h(\lambda),\begin{pmatrix}x\\y\end{pmatrix}) = WT(\lambda',\begin{pmatrix}x\\y\end{pmatrix})$$

$$\Leftarrow \{ \text{A sufficient condition} \}$$

$$\forall\lambda\exists\lambda': \lambda' = W_h(\lambda)$$

$$\Leftrightarrow \{ \ WT \text{ is a total function} \}$$

$$\texttt{true}$$

Repeating the same learning data does not create contradiction.

- *Convergence produces monotonicity.* We interpret convergence to be the situation where the new learning pair does not cause any change to the vector of weights. Formally,

  $$WT\left(\begin{pmatrix}w_1\\w_2\end{pmatrix},\begin{pmatrix}x\\y\end{pmatrix}\right) = \begin{pmatrix}w_1\\w_2\end{pmatrix}.$$

  Under this hypothesis, the condition of monotonicity

  $$\forall\lambda\exists\lambda': F^{\lambda}_{h.(x,y)} \ = \ F^{\lambda'}_{h}$$

  holds vacuously for $\lambda' = \lambda$. The idempotence of WT holds in particular when the learning process has converged (for the submitted learning data). Also, the formula of WT for our sample example provides that we have idempotence whenever the learning pair (x, y) satisfies the conditions:

  $$x = 0, y = \sigma(\frac{w_2}{2}).$$

6. CONCLUSION

On-line learning systems in general, and their neural net implementations in particular are gaining increasing acceptance in control applications, which are often characterized by complexity and criticality. A significant obstacle to their acceptance and usefulness/ usability is the lack of adequate verification/certification methods and techniques, as all traditional methods and techniques are inapplicable. In this paper we are presenting a tentative computational model for on-line learning systems and we use this model to sketch verification methods. Among the main contributions of our work, we cite:

- An abstract computational model that captures the functional properties of an evolving adaptive system by abstracting away random factors in the function of the system, to focus exclusively on details that are relevant to the learning algorithm and the learning data.
- The integration of this computational model into a refinement logic, which establishes functional properties of adaptive systems using refinement-based reasoning.
- The introduction of two venues for verifying adaptive systems: one based on *monotonic learning* (the *adaptive* equivalent of testing), and one based on *safe learning* (the *adaptive* equivalent of proving).
- The introduction of a (sketchy, so far) framework for inductive reasoning on adaptive systems; this framework is based on the proposed computational model, and aims to support the *adaptive* equivalent of the inductive methods of program proving.
- Some preliminary exploration of monotonic learning, whereby we provide sufficient conditions for monotonic learning, discuss them, and illustrate them.

While this work is still in its infancy, we feel that it has introduced some meaningful concepts and has opened original venues for further exploration, by taking a refinement-based approach.We envisage the following extensions to this work:

- Experiment, be it on small examples, with the derivation of the functional envelope (R_h) of the system, and analyze what the conditions of monotonic learning and safe learning mean in practice. While it is easy to compute relation R_h extensionally, by listing some of its pairs (as we have done in figure 4), it is not trivial to derive a closed form expression of it.
- Investigate means to obviate the need to derive an explicit closed form expression for R_h, by exploring inductive arguments that allow us to ensure monotonic learning and safe learning without computing the functional envelope.
- Fine-tune the proposed computational model and investigate its applicability to other forms of on-line learning systems (other than the back-propagation algorithm).
- Derive tighter sufficient conditions for monotonicity, and further analyze the condition of safe learning.
- Explore inductive proof methods for adaptive systems, along the lines of the framework proposed in this paper.

This research is currently under way. Cutting across all the research directions is the issue of scaling up; we are interested in exploring how our proposed solutions can scale up to realistic applications. Whereas traditional methods of verification of neural nets produce statistical results, our long term goal, in this work, is to produce logical claims we can make about the current and future behaviour of the system.

NOTES

* This work is funded by grants from NASA Dryden Flight Research Center, through the Institute of Software Research (Fairmont, WV), and from NASA Goddard Space Flight Center, through NASA IV\&V Facility (Fairmont, WV).

REFERENCES

[1] J.R. Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] Ch. Alexander, D. DelGobbo, V. Cortellessa, A. Mili, and M. Napolitano. Modeling the fault tolerant capability of a flight control system: An exercise in SCR specifications. In *Proceedings, Langley Formal Methods Conference*, Hampton, VA, June 2000.

[3] H. Ammar, B. Cukic, C. Fuhrman, and Mili. A comparative analysis of hardware and software fault tolerance: Impact on software reliability engineering. *Annals of Software Engineering*, 10, 2000.

[4] A. Avizienis. The n-version approach to fault tolerant software. *IEEE Trans. on Software Engineering*, 11(12), December 1985.

[5] R.J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer Verlag, 1998.

[6] N. Boudriga, F. Elloumi, and A. Mili. The lattice of specifications: Applications to a specification methodology. *Formal Aspects of Computing*, 4:544--571, 1992.

[7] M. A. Boyd, J. Schumann, G. Brat, D. Giannakopoulou, B. Cukic, and A. Mili. Ifcs project: Validation and verification (v&v) process guide for software and neural nets. Technical report, NASA Ames Research Center, September 2001.

[8] Ch. Brink, W. Kahl, and G. Schmidt. *Relational Methods in Computer Science*. Springer Verlag, New York, NY and Heidelberg, Germany, 1997.

[9] Rodney A. Canfield. Meet and join within the lattice of set partitions. Technical report, The University of Georgia, Athens, 2001.

[10] M. Caudill. Driving solo. *AI Expert*, pages 26--30, September 1991.

[11] C. H. Dagli, S. Lammers, and M. Vellanki. Intelligent scheduling in manufacturing using neural networks. *Journal of Neural Network Computing*, pages 4--10, 1991.

[12] J. Dean. Timing the testing of cots software products. In *First International ICSE Workshop on Testing Distributed Component Based Systems*, Los Angeles, CA, May 1999.

[13] Jules Desharnais, Ali Mili, and Thanh Tung Nguyen. Refinement and demonic semantics. In Brink et al. [8], chapter 11, pages 166--183.

[14] E.W. Dijkstra, *A Discipline of Programming*. Prentice Hall, 1976.

[15] D. Bernard et al. Final report on the remote agent experiment. In *NMP DS-1 Technology Validation Symposium*, Pasadena, CA, February 2000.

[16] B. Fritzke. Growing self-organizing networks - why. In *European Symposium on Artificial Neural Networks*, pages 61--72, Brussels, Belgium, 1996.

[17] LiMin Fu. *Neural Networks in Computer Intelligence*. McGraw Hill, 1994.

[18] P. Gardiner and C.C. Morgan. Data refinement of predicate transformers. *Theoretical Computer Science*, 87:143--162, 1991.

[19] D. Del Gobbo and B. Cukic. Validating on-line neural networks. Technical report, Lane Department of Computer Science and Electrical Engineering, West Virginia University, December 2001.

[20] D. Gries. *The Science of programming*. Springer Verlag, 1981.

[21] H. Hecht, M. Hecht, and D. Wallace. Toward more effective testing for high assurance systems. In *Proceedings of the 2nd IEEE High Assurance Systems Engineering Workshop*, Washington, D.C., August 1997.

[22] Internet. Program verification system. Technical report, SRI International Computer Science Laboratory, 1997.

[23] J. A. Leonard, M. A. Kramer, and L. H. Ungar. Using radial basis functions to approximate a function and its error bounds. *IEEE Transactions on Neural Networks*, 3(4):624--627, July 1991.

[24] M. Lowry, M. Boyd, and D. Kulkarni. Towards a theory for integration of mathematical verification and empirical testing. In *Proceedings, 13th IEEE International Conference on Automated Software Engineering*, pages 322--331, Honolulu, HI, October 1998. IEEE Computer Society.

[25] Z. Manna. *A Mathematical Theory of Computation*. McGraw Hill, 1974.

[26] H.D. Mills, V.R. Basili, J.D. Gannon, and D.R. Hamlet. *Structured Programming: A Mathematical Approach*. Allyn and Bacon, Boston, Ma, 1986.

[27] C.C. Morgan. *Programming from Specifications*. International Series in Computer Sciences. Prentice Hall, London, UK, 1998.

[28] M. Napolitano, G. Molinaro, M. Innocenti, and D. Martinelli. A complete hardware package for a fault tolerant flight control system using on-line learning neural networks. *IEEE Control Systems Technology*, January 1998.

[29] M. Napolitano, C. D. Neppach, V. Casdorph, S. Naylor, M. Innocenti, and G Silvestri. A neural network-based scheme for sensor failure detection, identification and accomodation. *AIAA Journal of Control and Dynamics*, 18(6):1280--1286, 1995.

[30] D.B. Parker. Learning logic. Technical Report S81-64, Stanford University, 1982.

[31] G. E. Peterson. A foundation for neural network verification and validation. *SPIE Science of Artificial Neural Networks II*, 1966:196--207, 1993.

[32] D. K. Pradhan. *Fault Tolerant Computing: Theory and Practice*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[33] B. Randall. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2), 1975.

[34] Orna Raz. Validation of online artificial neural networks ---an informal classification of related approaches. Technical report, NASA Ames Research Center, Moffet Field, CA, 2000.

[35] D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume I: Foundations*. MIT Press, Cambridge, MA, 1986.

[36] D.P. Siewiorek and R. S. Swarz. *The Theory and Practice of Reliable System Design*. Digital Press, Bedford, Mass, 1982.

[37] Boeing Staff. Intelligent flight control: Advanced concept program. Technical report, The Boeing Company, 1999.

[38] P.J. Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. Technical report, Harvard University, 1974.

[39] J. Von Wright. A lattice theoretical basis for program refinement. Technical report, Dept. of Computer Science, Abo Akademi, Finland, 990.

Experimenting with Genetic Algorithms to Devise Optimal Integration Test Orders

Lionel C. Briand, Jie Feng and Yvan Labiche

Software Quality Engineering Laboratory
Carleton University
Department of Systems and Computer Engineering
1125 Colonel By Drive
Ottawa, ON, K1S 5B6, Canada
{briand, labiche}@sce.carleton.ca

ABSTRACT

We present here an improved strategy to devise optimal integration test orders in object-oriented systems in the presence of dependency cycles. Our goal is to minimize the complexity of stubbing during integration testing as this has been shown to be a major source of expenditure. Our strategy to do so is based on the combined use of inter-class coupling measurement and genetic algorithms. The former is used to assess the complexity of stubs (each coupling measure capturing a dimension of this complexity) and the latter is used to minimize cost functions based on coupling measurement. Using a precisely defined procedure, we investigate this approach in a case study involving five real systems. Results are very encouraging as the approach clearly helps obtaining systematic results that are close to be minimal in terms of stubbing complexity.

1. INTRODUCTION

One important problem when integrating and testing object-oriented software is to decide the order of class integration [3]. A number of papers have provided strategies and algorithms to derive an integration and test order from dependencies among classes in a system [6, 17, 18, 21, 24]. The objective of all these approaches is to minimize the number of test stubs to be produced, as this is perceived to be a major cost factor of integration testing. Stubs are software units that are necessary to run the software under test (i.e., it depends on them) and that must be developed as part of the test harness along with drivers and oracles [2]. Such a need stems from the fact that, in most software development projects, components are developed and tested concurrently by different developers and integration begins before the component development and testing phase are complete.

One specific issue is how to deal with dependency cycles, which prevent any topological ordering of classes. This is important as class diagrams (e.g., UML class diagrams defined during Analysis or Design, or reverse-engineered) of most object-oriented software systems contain dependency cycles. All the solutions proposed in [6, 21, 24] are based on the principle of "breaking" some dependencies to obtain acyclic dependencies between classes. A broken dependency implies that the target class will have to be stubbed when integrating and testing the source class. Furthermore, all these solutions are based on search algorithms in directed graph representations of class dependencies, and we discuss below what the limitations and advantages of such an approach are.

A first attempt has been made to use Genetic Algorithms (GA's) to address the test order issue [20]. GA's are a family of global optimization techniques based on heuristics and developed by the artificial intelligence community [8]. In [20], the authors report experiments on six different systems (4 of which are libraries) that yield overall poorer results with GA's when compared with their graph-based approach. However, little information and justification regarding the settings used for the GA (e.g., cross-over and mutation rate values, population sizes) are provided and, considering that these algorithms are known to be sensitive to such parameters, it is difficult to generalize or conclude from these results. Furthermore, the inherent uncertainty related to GA's (which are based on heuristics) is not investigated and little insight is given on some of the most unexpected results (e.g., in the Java library, 7 stubs break 8 thousand cycles!). We thus believe that a rigorous, scientific investigation of the use of GA's is necessary, thus precisely characterizing their strengths and limitations, and the conditions under which they are a useful alternative.

In practice, we would like to refine graph-based algorithms in order, for example, to account for the complexity of dependencies between classes. In other words, different dependencies that we consider breaking may lead to stubs of widely varying complexity. Such complexity is driven by the coupling that exists between the client and server class of a dependency. If the client uses a lot of features of the server, then breaking the dependency is expected to lead to an expensive stub. Given our objective, modifying graph-based algorithms to account for stub complexity based on coupling measurement turns out to be very complex and leads to intractable algorithms.

In order to address this problem, and for reasons that are described below, we then turned our attention to Genetic Algorithms (GA's). As further discussed below, the main motivation for using these algorithms is their flexibility and practicality in using a large range of optimization problems. Their popularity in software engineering is growing as many of the problems we face can be re-expressed as optimization problems. This is further illustrated by a recent journal issue dedicated to the applications of metaheuristics algorithms to software engineering [14]. This paper focuses on the best ways to use GA's in order to devise optimal test orders based on dependency coupling measurement between client and server classes.

Last, there exist other integration strategies that are not based on class diagram information, as derived from the software design or reverse-engineering. They rather associate a functional description with (a set of) classes. For instance, in [15], Atomic System Functions (ASF), which involve system inputs and outputs, and exercise Method/Message paths between objects, drive the integration test of classes. These ASFs correspond to a functional decomposition of the system, which is similar to use cases. The objective of the strategy is not to minimize test stubs but to execute complete, end-user functionalities, in an incremental manner during integration. Similar strategies using use cases can be found in [3, 22]. Since these strategies are not explicitly based of the class diagram, they will not be detailed and compared in this article. Though this is a topic of future research, it is very likely that in practice, those two sets of strategies would have to be combined.

We provide motivations and a methodology in Section 2. Section 3 then provides the detailed results of five case studies where we demonstrate that the approach is not only feasible but also yields low stubbing complexity test orders. Section 4 concludes and outlines future research.

2. APPLYING GENETIC ALGORITHMS TO THE TEST ORDER PROBLEM

We first provide some motivations that lead us to select Genetic Algorithms over alternative approaches (Section 2.1). We then go into presenting an overview of the relevant, fundamental principles we will be using in the remainder of the paper (Section 2.2). We then go into describing how we tailored Genetic Algorithms to our specific problem and report a procedure to use them to devise low stubbing complexity test orders (Section 2.3).

2.1. Motivations

We first discuss the integration order problem in more depth. We then discuss why it is difficult to adapt existing graph-based algorithms to accommodate our needs in determining test orders and then justify why Genetic Algorithms seem to be a good alternative to investigate.

2.1.1. The Integration Order Problem

Once classes have been developed and tested in isolation, one important problem is to integrate them into a (sub)system. Due to the usual problems associated with big-bang integration, classes need to be integrated one at a time or, in some cases, in small clusters. A practical question is now to define precisely what an optimal integration order actually is. We need to define evaluation criteria to compare orders and find algorithms that can identify (near) optimal orders automatically.

A natural evaluation criterion is the stubbing effort required to integrate classes according to a specific order. If an order integrates classes when some of the classes they depend on are not yet integrated, the development of stubs to substitute for those classes is necessary to perform integration testing. It has been observed that such stubbing is error-prone and significantly increases the cost of integration testing [2].

Existing algorithms that devise class test orders are typically based on analyzing dependencies between classes [6, 16, 18, 21, 24]. If there are no cycles of dependencies among classes, the problem becomes a simple topological sorting issue, a well-known graph theory problem [11], and it is then possible to obtain integration orders that do not require the stubbing of any class. In the case where dependency cycles can be observed, there is not choice but to integrate some classes when certain classes they depend on are not yet integrated. This leads to the development of stubs and significantly increases the cost of integration testing. In such a situation, we need to devise integration orders that *minimize* stubbing effort. Since such effort cannot be directly measured or estimated, we need to resort to indicators.

Another important point to mention is that, though we focus on ordering class integration in this paper, the method we investigate can be applied at higher levels of integration. For example, in a very large system, we can easily imagine two or more levels of integration. A first level is concerned with integrating classes into subsystems, within each sub-team of the project. The second level would then address the integration of subsystems into a complete system. In large, complex systems, the dependencies among subsystems may be complex and show cycles. Those dependencies are determined by the public classes within subsystems, which form their public interface to other subsystems.

2.1.2. Graph-Based Solutions are not Satisfactory

Our previous work considered the number of stubs as an estimator for the cost associated with the construction of stubs, that is the number of dependencies to break so as to obtain an acyclic class dependency graph, and then applying topological sorting to obtain an integration test order. As a consequence, the problem was a graph-searching problem and was solved using typical graph algorithms. Basically, the proposed solutions in [6] and [21] consist in recursively identifying strongly connected components (SCC) and then, in each SCC, removing *one* dependency that maximizes the number of broken cycles. These solutions consider two distinct but simple cost functions to select the dependency to break at each recursion step, but they share the commonality of always optimizing their choice of dependencies to break at a given stage of recursion, without determining the consequences on the ultimate results (i.e., they are both a greedy algorithm). We will see next how this becomes a serious limitation in the context we stated in Section 1.

Let us first recall our needs in terms of devising optimal integration orders. We already mentioned the coupling associated with client-server dependencies, for which we can provide several complementary and alternative measures, e.g., number of calls made by the client class to the server class, number of distinct methods invoked. If such coupling is accounted for, we may find ourselves in a situation where, for instance, breaking two dependencies has a lower cost (i.e., coupling) than breaking only one dependency that would make the graph acyclic in one step. Furthermore, we would like to consider constraints, due to organizational or contractual reasons, which would then result into some classes not being available (developed and tested) before others during integration. In practice, such situations constrain the optimal ordering solution we are searching.

Given the above requirements, adapting the previous graph-based, stepwise strategies seems difficult or even impossible. The cost functions we are minimizing here (see Section 2.3.2) are highly non-linear and are likely to present many local minima in which a stepwise algorithm may get stuck[1]. Consequently, using coupling information in the cost function would require the identification, in each SCC, of all the *subsets* of breakable dependencies (e.g., pairs, triplets) that would make the graph acyclic and the determination of their associated cost to select an optimal solution. This is not a viable solution for any realistic class diagram.

Similarly, when using a stepwise search algorithm, it is difficult to determine whether breaking a dependency at a given step will not lead to orders, at subsequent steps, that will transgress some of the constraints which limit our search.

2.1.3. Moving to Genetic Algorithms

The solution space to our problem is the set of all possible test orders for a given system. So our solution is represented as an order (or at least a *partial* order) of classes and optimization means, in our context, searching for an order that minimizes a cost function. This cost function will be described in detail below but, based on our objectives, it is clear that it will be based on measuring dependencies/coupling between client and server classes and will consist in minimizing such coupling while breaking dependencies to remove cycles. In short our problem is a *constrained*, *multi-objective* optimization problem as it may involve a number of coupling measures.

Genetic algorithms (GA's) are a specific type of optimization techniques that are based on a set of heuristics and involve a non-trivial and careful setting of a number of parameters. The basic principles that are relevant to our problem will be summarized in the next section. In our context, one of their interesting characteristics is that they prevent the optimization search from getting stuck into local minima and this is why they are often referred to as *global* optimization techniques. GA's are also known to help solve complex, non-linear[2] problems that often lead to cases where the search space shows a curvy, noisy "landscape" with numerous local minima [14]. However, as they are based on heuristics there is no guarantee they will find the absolute, global minimum.

A category of GA's is designed to address so-called routing or scheduling problems [8], which is very much related to our integration test order problem. A typical example is the Traveling Salesperson Problem (TSP) where N towns are distributed around a two dimensional Euclidian space. The salesperson is assumed to traverse all the N towns, starting with an initial one and getting back to it in the end. The problem is to minimize the distance (cost function) covered by the salesperson while achieving this objective. The representation of the problem is also an order, more precisely a sequence of towns being traversed. Though there are significant differences between our problem and the TSP, we can see that it is analogous since it looks for an optimal order minimizing a cost function. Furthermore, reported examples of scheduling and routing problems (e.g., [28]) suggest that GA's provide satisfactory results. The most important differences between our integration order problem and the TSP problem mainly stems from our cost function:

- In the TSP, if the order places A before B or vice-versa, the cost is the same as it is the distance between the two towns. This is not the case in our context where one solution may entail a stub whereas the other one does not.
- Similarly, since the distance is the cost in the TSP, the triangle inequality holds where the distance of ABC is equal or larger than that of AC. Such inequality has no reason to hold in our context.

Those differences could have significant impact on the ability of the GA to converge towards an optimal solution and this further justifies the need to perform case studies to assess the ability of GA's to solve the integration test order problem. Another difference with the TSP problem is the fact that we will need to find an optimal solution under constraints specifying whether orders are acceptable or not. Section 2.3.3 shows how this can be practically achieved with GA's. Furthermore, it is interesting to note that, though deterministic algorithms to compute near-optimal tours in the TSP problem have been proposed [19], the differences above prevent us from using them.

Another practical motivation to use GA's is that commercial tools have reached a high level of maturity and provide convenient features for solving real-scale problems. This is further discussed in Section 3.1.

2.2. Fundamental Principles and Application to Scheduling Problems

A Genetic Algorithm (GA) is an optimization technique that has the ability to find a *global* optimum, and avoid getting stuck in a *local* optimum. However, finding the global optimum is not guaranteed as GA's are based on heuristics. Therefore, for each new problem to be solved, it is necessary to investigate empirically how GA's perform in representative situations. A GA allows an initial population composed of many solutions to the problem stated (called *chromosomes*) to evolve under specified selection rules to a state that optimizes (say, minimizes for the rest of this discussion) a cost function. The parameters involved in the cost function one wants to minimize are first encoded into a chromosome (e.g., a chromosome can be seen as an ordered list of parameter values). There are several possible *encoding* strategies to specify how a solution is represented and their adequacy depends on the problem under consideration (e.g., continuous function optimization or order optimization). In all cases a chromosome is composed of a number of *genes*. When the initial population evolves from one generation to the other, the best chromosomes are preserved whereas the others are discarded to make room for the new offspring(s). This emulates natural selection that keeps the best-fitted individuals. New offsprings are produced using an evolution operator, named *crossover*, so as to keep the same number of individuals in the next generation. A crossover produces two new chromosomes that share part of the genes coming from two of the best chromosomes in the original population. Another operator, called mutation, affects the population by mutating some of the genes of the chromosomes. These operators allow the algorithm to leave the area, in the solution space, of a minimum that may turn out to be local. Though the overall principles make sense, a number of questions arise in practice:

- What is an adequate encoding for the problem?
- What are possible crossover and mutation operators, which are adapted to the selected encoding?
- What is the rationale for the use of specific crossover and mutation operators?
- What should be the size of the initial population?
- What should be the number of best-fitted chromosomes kept from one generation to the other?
- How do we consider we have obtained an acceptable solution?

Let us now consider our integration test problem and answer these questions in context. Recall we need to determine an order of class integration that is optimal in the sense that it minimizes stubbing complexity (our cost function). The first step is to decide how to encode the solution to our problem. It seems natural in our case to select a *permutation* encoding: Every chromosome is a string of class labels. For a set of classes {A, B, C, D}, a possible chromosome is the sequence (B C A D): The population from which new generations are generated is a set of such chromosomes, e.g., {(B C A D), (B C D A), …}.

Recall that in the context of GA's, *crossover* and *mutation* operators need to be performed on the original population and subsequent generations until we believe we have obtained an acceptable (possibly optimal) solution. Those operators are specific to permutation encoding as they need to ensure that correct sequences are produced, that is sequences with strictly one occurrence of each class. Mutation is implemented by selecting two classes and swapping their positions in the chromosome. For example, highlighting a randomly selected pair of genes (classes in our test order), (B **C** **D** A) is mutated into (B **D** **C** A). Crossover is a more complex operator and several crossover operators have been proposed in the literature for permutation encoding [13]. One of them, which is used in our study, is described in [9] and works as follows: Genes (classes in our context) are randomly selected from a first parent chromosome (test order), their places are found in the other parent, and the remaining genes are copied into the first parent in the same order as they appear in the second parent. This preserves some of the sub-orderings in the original parents while creating some new sub-orderings. For example, if we assume the following two parent chromosomes:

Parent 1: (A B C D)

Parent 2: (D C B A)

If we further assume that A and C are selected in the first parent, we produce a first child chromosome as follows:

- We obtain (A - C -), where A and C have been selected and ‘-‘ are gaps;
- We fill in the gaps with B and D in the order they are encountered in parent 2 and obtain (A D C B).

GA's also require that a number of parameters be set. The first one is the *population size*. It has an impact on the speed of the GA convergence towards an optimal solution and its capability to avoid local optima. Larger population sizes increase the amount of variation present in the initial population at the expense of requiring more cost function evaluations and longer execution times. Typical population sizes in the literature range between 25 and 100 [1]. However, for longer chromosomes and challenging optimization problems, larger population sizes are needed to ensure diversity among the chromosomes and hence allow a more thorough exploration of the solution space. In the context of integration orders, the population size will consequently be driven by the number of classes in the class diagram. As a heuristic, having a population size of two or three times the number of classes should be sufficient.

Another parameter is the crossover rate, that is the probability that a chromosome will undergo a crossover. Typical rates in the literature for the TSP range from 0.5 to 0.6, consistent with De Jong's simulation results [10]. He reported that, on a number of different optimization problems, simulation results suggested that a crossover rate of roughly 0.6 was a good compromise between a number of performance measures.

Mutation prevents the GA search to fall into local minima, but they should not happen too often or the search will converge towards a random search. The mutation rate is defined as the probability for a chromosome to undergo a mutation. Typically, for the TSP, the literature reports rates around 0.15. Recent theoretical work provides a rule of thumb of 1/N, where N is the number of genes in the chromosomes [1]. [27] argue that for more complex encoding, such as order encoding, high mutation rates can be both desirable and necessary.

2.3. Tailoring and Application Procedure

In this section, we present how the Genetic Algorithm was parameterized based on the specifics of our study and results reported in the literature. We then define what cost functions will be minimized and how we proceed to a priori restrain our search space.

2.3.1. Parameter Settings

We selected a mutation rate of 0.15, consistent with the literature on the TSP. We also tried 0.05 and 0.1, to be more in line with the 1/N heuristic stated above, but it did not make any significant difference. Regarding the crossover rate, we tried both 0.5 and 0.6, noticing no significant difference and decided to pursue the experiments with 0.5. Our population is made of 100 chromosomes, this being the largest population size in most case studies in the literature and far larger than the number of genes in the chromosomes of our case study (Section 3).

Last, we allow for 500 trials to converge towards an optimal result. Preliminary experiments using larger numbers showed that this was sufficient to converge, thus saving us significant computation time. A related point to mention is that we use steady-state replacement strategy [9]: A large proportion of the chromosomes in a population should survive to the next generation. The underlying rational is that we should ensure we preserve the best-found solutions in the next generation. In our particular case, one chromosome (the worst one) gets replaced at each trial after two parent chromosomes are selected (the best-fitted chromosomes are more likely to be selected) and an offspring is generated by performing crossover and mutation operations on them, as described in Section 2.2. Note that we use the term *best-found* to denote the best result obtained with GA's. Since they make use of search heuristics, there is no guarantee to obtain the *optimal* order, which remains unknown. We also sometimes refer to *sub-optimal* orders to mean results that are less satisfactory than the best-found order.

2.3.2. Measuring Stubbing Complexity

A number of dependencies can be found between classes in a UML class diagram. Compositions and inheritance relationships are considered unbreakable in our strategy as, according to our heuristic, breaking them would likely lead to complex stubs. Such relationships usually entail tight dependencies between the client/parent and server/child classes [6]. For remaining dependencies, that is associations, simple aggregations, and usage dependencies, we compute a complexity based on the level of coupling they involve. In this paper, we measure coupling in two simple, intuitive ways:

A(Dependency): The number of attributes *locally* declared[3] in the target class when references/pointers to instances of the target class appear in the argument list of some methods in the source class, as the type of their return value, in the list of attributes (data members) of the source class, or as local parameters of methods. This complexity measure counts the (maximum) number of attributes that would have to be handled in the stub if the dependency were broken.

M(Dependency): The number of methods (including constructors) *locally* declared[3] in the target class which are invoked by the source class methods (including constructors). This complexity measure counts the number of methods that would have to be emulated in the stub if the dependency were broken. Note that this is an approximation as some of the methods can be overridden.

More measures could be defined in the future [4], but those two measures are enough to illustrate the benefits of our approach. Based on the two measures *A()* and *M()*, we can then define a cost function to be minimized by the GA. We define the complexity of a dependency as being the geometric average of all complexity measures, i.e., *A()* and *M()* in the current example. However, before we compute such an average we need to make sure the resulting complexity is not sensitive to the measurement units of each complexity measures. For examples, if the number of methods tend to be on average higher than the number of attributes, *M()* will have a larger weight than *A()* in the total complexity of a dependency, though this was not originally intended. To address this issue, we normalize *A()* and *M()* so that their range is between 0 and 1. In mathematical terms, for a measure *Cplx()*, we compute its corresponding *normalized* measure $\overline{Cplx}()$. If we assume that complexity information is represented as a matrix *Cplx(i,j)* where rows and columns are classes and i depends on j, we have to compute Cplx$_{\mathrm{min}}$ = Min{Cplx (i, j), i, j = 1, 2, …} and Cplx$_{\mathrm{max}}$ = Max{Cplx (i, j), i, j = 1, 2, …..} and then perform the following computation:

$$\overline{Cplx(i,j)} = {}^{Cplx(i,j)}\Big/_{Cplx_{\mathrm{max}} - Cplx_{\mathrm{min}}}$$

For the two complexity measures we define here, the minimum of complexity value is 0 and the equation can be simplified:

$$\overline{Cplx(i,j)} = {}^{Cplx(i,j)}\Big/_{Cplx_{\mathrm{max}}}$$

Then, based on *A()* and *M()*, the overall stubbing complexity *SCplx(i,j)*, for a dependency linking a pair of classes *(i,j)*, can be computed as a weighted geometric average of the normalized measures:

$$SCplx(i,j) = (W_A \cdot \overline{A}(i,j)^2 + W_M \cdot \overline{M}(i,j)^2)^{1/2}$$

where W_A and W_M are weights and $W_A+W_M=1$.

For a given test order *o*, a set of *d* dependencies (i.e., denoted above as pairs of classes) to be broken is identified and an overall complexity can be computed for the order as:

$$OCplx(o) = \sum_{k=1}^{d} SCplx(k)$$

The measure *OCplx()* is the *cost function* we try to minimize by using GA. Fully defining such a cost function requires the determination of weights for all complexity measures involved, e.g., W_A and W_M. On the one hand, those weights allows the user to tailor the cost function to its own intuitions about the kinds of complexity that have more bearing on stubbing effort. On the other hand, this is difficult as setting such weights is a subjective task. This is further discussed in Section 3.4 based on the results presented in Section 3.3.

2.3.3. Constraints

Recall that, according to our strategy, *Inheritance* and *Composition* dependencies cannot be broken. This means the base/container classes must precede child/contained classes in any order that is generated by the GA. In other words, we must optimize the integration order under certain constraints. As we will see, many GA tools such as Evolver [23] allow for the specification of a *precedence table* where the conditions for a new offspring to be acceptable are specified under the form of a partial order. If after applying crossover and mutation operators to selected parent chromosomes an offspring does not fulfill the constraints in the precedence table, the tool backtracks and a new offspring is re-generated until it conforms with the precedence table. The details of such backtracking algorithms vary from tool to tool and are usually proprietary.

Another important set of constraints comes from the fact that we only accept breaking *Association* and *Usage* dependencies that are part of at least one dependency cycle. Integration orders that do not fulfill such constraints are by definition suboptimal. In our previous work where graph-based approaches were explored for optimizing test orders [6], we used Tarjan's algorithm [25] to detect strongly connected components (SCC's) in the graph formed by classes and their dependencies. This algorithm can be used again here to detect SCC's and only dependencies between classes part of at least one SCC should be considered for breaking by the GA. While this type of constraints is in theory not required by the GA to converge towards optimal orders we have observed that the speed of convergence significantly increased by doing so (less generations and execution time). The constraints turned out to be a very effective way to constrain the search space and improve the effectiveness of the GA heuristics.

Another interesting source of constraints comes from the use of design patterns in the class diagram. Let us take as an example the state design pattern [12], which is typically used to design classes with a state dependent behavior. The *context* class, whose state is being modeled, is related through an aggregation relationship to an abstract class *State* (see Figure 1). Every time the context class receives a message, its state is likely to change, thus requiring a message being sent to an instance of one of the subclasses of *State*. In turn, the subclasses instances may invoke *action* methods (e.g., *Action1*) in the context class. In this situation, even if the aggregation is a candidate for breaking when involved in a cycle, it does not seem reasonable to separate the test of classes *Context* and *State* during integration testing. Though this needs to be further studied, similar conclusion can be drawn when using other design patterns.

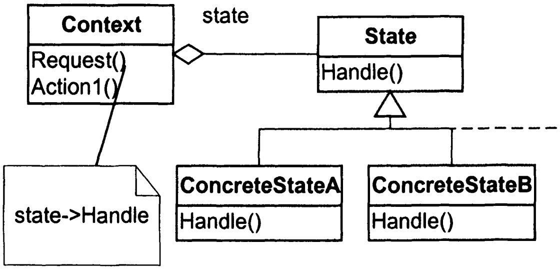


Figure 1. State Design Pattern.

In practice, yet another source of constraints stems from the availability of personnel or other development resources (e.g., off-the-shelf or outsourced software). The integration order is going to drive the development order as classes and subsystems need to be ready and tested when they are to be integrated. For example, though according to some optimal integration order some classes need to be developed before others, no personnel with suitable skills may be available and, therefore, such an integration order is simply not applicable. So it is important that such practical considerations be taken into account when finding optimal integration orders as they may play a key role in determining their feasibility.

All constraints, regardless of their source, may easily be accounted for when using GA tools through the use of precedence table and backtracking mechanisms.

3. CASE STUDIES

We describe in the first subsection our five case studies. Due to space constraints, details on how we reverse-engineered the class dependencies, and the coupling values, from the corresponding Java source code are not included in this article. The reader is invited to read the detailed descriptions provided in [5]. Such detailed information regarding the reverse-engineering of the ORDs (e.g., how to identify usages) was not provided in [20], thus preventing us from comparing their results regarding the use of Genetic Algorithms with ours. Results are then reported in the next subsection. For the five case studies, class diagrams and corresponding relationship tables, dependency matrices, precedence tables, coupling matrices, as well as some sample orders for each cost function on which are based the results below are not reported here but can be found in [5].

3.1. Design of Case Studies

The first system is an Automated Teller Machine (ATM) simulation (the classes connected to hardware devices are missing). The class diagram is made of 21 classes and 67 relationships, and contains 30 cycles[4] involving 8 of the 21 classes. The second system, named Ant, is part of the Jakarta project (http://jakarta.apache.org). Ant creates and maintains open source solutions on the Java platform for distribution to the public at no charge. The Ant system is a Java based build tool similar to the make tool on Unix platforms: It maintains, updates, and regenerates related programs and files according to their dependencies (e.g., compilation units). The class diagram consists in 25 classes and 83 relationships, and contains 654 cycles involving 12 of the 25 classes. The third example, named SPM (*Security Patrol Monitoring*), is a course project implemented by a graduate student at Carleton University. This system monitors security zones (e.g., authorized entry/exit) and patrols (e.g., schedules). The class diagram consists in 19 classes and 72 relationships, and contains 1,178 cycles involving 15 out of the 19 classes. The fourth example, BCEL (*Byte Code Engineering Library),* also comes from a subproject of Jakarta Project, and is intended to give users a convenient tool to analyze, create, and manipulate binary Java class files. We used the *org.apache.bcel.classfile* package of version 5.0 as our example (http://jakarta.apache.org/bcel/index.html*).* The class diagram is made of 45 classes, and 294 relationships, and contains 416, 091 cycles involving 41 out of 45 classes. The last application system, named dnsjava or simply DNS in this article, is an implementation of Domain Naming System in Java: i.e., it provides network naming services (http://www.xbill.org/dnsjava/). The DNS class diagram consists in 61 classes and 276 relationships, and contains 16 cycles involving 10 out of 61 classes.

These five application systems[5] were chosen because they were deemed to be of sufficient size and of varying complexity, so as to assess the effectiveness of the GA-based approach. ATM, Ant, SPM, and BCEL have class diagrams of reasonable (and comparable) sizes (between 19 and 45), but with very different numbers of cycles (from 30 for ATM to 416,091 for BCEL). On the other hand, the DNS system has the most important number of classes (and almost the same number of relationships as BCEL), but the smallest number of cycles (fewer number than ATM, Ant, and SPM)! This shows the topography of class diagrams can vary a great deal across application systems. Further details about the systems are provided in Table 1.

System	Usages	Associations & Aggregations	Compositions	Inheritance
ATM	39	9	15	4
Ant	54	16	2	11
SPM	24	34	10	4
BCEL	18	226	4	46
DNS	211	23	12	30

System	Classes	Cycles	# LOC
ATM	21	30	1390
Ant	25	654	4093
SPM	19	1178	1198
BCEL	45	416, 091	3033
DNS	61	16	6710

Table 1. Detailed Information for the Five Case Studies.

Figure 2 provides coupling distributions, for both our attribute and method measures, for the five case studies, under the form of histograms: the y-axis indicates the number of breakable relationships (i.e., UML associations, aggregations and use dependencies) that have the (attribute or method) coupling value indicated on the x-axis. These coupling distributions are summarized in Table 2 by means of ranges and average values.

	Attribute Coupling		Method Coupling	
	Range	Average	Range	Average
ATM	[1, 13]	6.15	[0, 7]	1.79
Ant	[0, 31]	8.36	[0, 14]	2.52
SPM	[1, 16]	7.53	[0, 8]	2.33
BCEL	[0, 20]	1.89	[0, 7]	1.52
DNS	[0, 12]	3.36	[0, 10]	1.46

Table 2. Coupling Summary for the Five Case Studies.

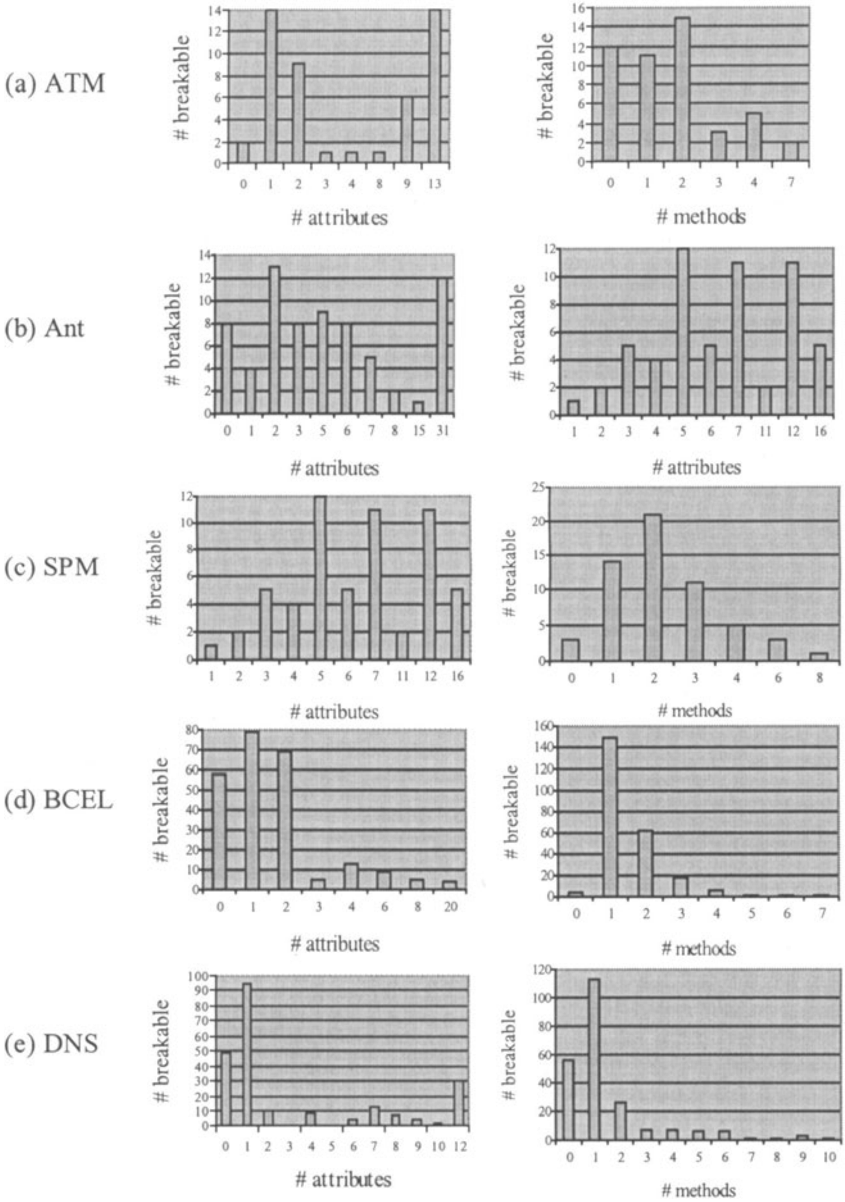


Figure 2. Coupling (Attribute and Method) Distributions for the Five Case Studies.

Though the number of classes involved in the 5 selected systems may seem modest by comparison with some of the systems that are commonly developed across the software industry, recall that in large systems the strategy we describe in this article would be used at different levels of integration, in a stepwise manner. For example, the GA algorithm would first be used to integrate classes into lower-level subsystems and then lower-level subsystems into higher-level subsystems, step by step until the system is entirely integrated. It is then unlikely that a given subsystem contains more than a couple hundred classes or more than a few dozens lower-level subsystems. Note that in the case where lower-level subsystems are integrated, the GA technique presented here are also required and used in the same manner, based on a dependency graph which nodes are subsystems and which dependencies are reflecting the dependencies of the classes they contain, focusing exclusively on those that cross subsystem boundaries.

3.2. The Application of Genetic Algorithms

The first question we want to investigate is whether GA's, as parameterized in this paper, work as well as graph algorithms such as the ones described and experimented with in [6, 7]. More precisely, we want to compare results produced by GA's with results produced by the graph-based solution described in [6], which has been shown to be the best graph-based approach in [7] (by means of analytical and empirical evaluations). So we define the cost function as the number of dependencies to be broken (*D*) in a given test order and run the GA 100 times. Since such an algorithm is a heuristic, the results may differ from run to run and we therefore check the percentage of times the algorithm converges towards the orders using the graph-based search algorithm. This question is important as it is a preliminary but necessary validation of the approach we describe in this paper.

The second question we investigate is related to the use of coupling measurement in the cost function (*OCplx*), as described in Section 2.3.2. We want to know whether such measurements make a significant difference in terms of the test orders that are generated, as compared to the orders we obtain with the graph-based algorithm. To answer this question, we compare the orders produced by each of the following cost functions:

- Only the number of broken dependencies is used as a cost function: SCplx(i,j) ∈ {0,1}, depending on whether *i* depends on *j* or not.
- Attribute and Method coupling are used in turn as the cost function: SCplx(i,j) = A(i,j), and SCplx(i,j)=M(i,j).
- A weighted geometric average of attribute and method coupling (*Ocplx*) is used in the definition of *SCplx*.

In other words, these four cost functions are used in turn to produce sets of orders on which we compute the values of the three other functions. If we observe *significant* differences in *OCplx* (order stubbed complexity) when only the number of dependencies is used as a cost function and when *OCplx* itself is used, then we can conclude that using coupling measurement may lead to significantly different test orders, involving less coupling and hopefully leading to lower integration test expenditures. Furthermore, if using *OCplx* as a cost function leads to test orders which are a reasonable compromise between the number of broken dependencies and their corresponding number of attributes and methods, then we can conclude that cost functions as defined in *OCplx* can be useful to achieve practical test orders in the context where one has multiple optimization objectives, as captured by the coupling measures. To address the latter question, we must compare the results obtained with a weighted geometric average and those obtained when using Attribute and Method coupling alone as cost functions. Such a comparison tells us whether a normalized, weighted average used as a cost function can achieve results that are close to the best-found values[6] of broken dependencies, Attribute coupling, and Method coupling.

Furthermore, since GA's are based on heuristics, it is important we investigate the reliability and repeatability of the results we obtain. Ideally, we would also like to know how far the minima we find are from the actual global minima, which are unknown. This is unfortunately impossible, unless we use simplistic examples. But in light of the complexity of the systems we use as case studies, we may be able to assess whether the results we obtain are plausible and likely to be close to actual global minima.

In order to automate our study, we used a commercial tool: Evolver 4.0 [23]. The choice of tool was made based on a number of criteria. First, we needed the tool to handle permutation encoding. Second we needed to be able to handle possibly large numbers of classes and we needed to make sure the number of genes in the chromosomes could be large enough as many tools have limitations with this respect. Similar requirements for the maximum size of populations had to be considered since when the number of genes grows, one needs to generate larger populations. Other practical features included the random generation of initial populations (in such a way that the diversity of chromosomes is ensured) and the graphical display of the average and minimum cost function value from generation to generation. The latter allows the user to stop the production of new generations when no noticeable improvements can be observed over a sequence of generations. Last, a feature that appeared more and more important over time was the possibility of specifying constraints defining acceptable orders and backtracking mechanisms to account for those constraints when generating new chromosomes.

3.3. Results

3.3.1. The ATM System

We ran the search algorithm 100 times with D as the cost function and found that, in *every* case, we obtain the best order found with our graph-based search algorithm [7]: 7 dependencies are broken and stubbed. This suggests that, under similar conditions, GA's can perform as well as deterministic algorithms. This result is of course dependent on the way we tailored and parameterized the algorithms.

Table 3 summarizes the overall results we obtain: Columns represent the different ways to measure stubbing complexity, in terms of broken dependencies and stubbed methods and attributes, whereas rows show the different cost functions that can be minimized using GA's, as described in Section 2.3. Highlighted cells show the cases where minimal *OCplx* values are obtained. In some cases, intervals are shown as, as expected, the results are not always consistent across the 100 executions of the GA. We also provide, below the interval, the mean and median values.

From Table 3, we can notice that the best result is consistently obtained for each cost function (no interval). When *D* is used as the cost function, the number of dependencies broken is the best (7), but this is not systematically the case for the values of *A*, *M*, and *OCplx* which are going up to 67, 19, and 4.18 instead of their best-found values of 39, 13, and 2.68, respectively[7]. Therefore, when using the number of dependencies broken as a cost function, we frequently obtain orders that are significantly suboptimal in terms of attributes and methods stubbed. As we will see next, this can be fixed by accounting for both *A* and *M* in the cost function, which leads to orders that are systematically the best. When the cost function is *M*, we systematically obtain the best-found value for *M* (13) and the number of dependencies, but not for *A* (and *OCplx*). When the cost function is *A*, we systematically obtain the best-found value for *A* (39) and the number of dependencies, but not for *M* (and *OCplx*).

When using *OCplx,* with $W_M=W_A=0.5$, as the cost function, we obtain best-found orders in terms of dependencies, methods, and attributes. Though we may have been lucky in this example, this result suggests that such cost functions, where all coupling measures are given an equal weight and a geometric average is computed, may be a useful, practical means to achieve a reasonable compromise among multiple objectives, as formalized by the coupling measures.

	Stubbing Complexity Values			
Cost Functions	D	A	M	OCplx
D	7	[39-67]	[13-19]	[2.68-4.18]
	-	53, 53	19, 16.72	3.44, 3.47
A (OCplx, W_A=1)	7	39	[13-19]	[2.68-2.98]
	-	-	19, 17	2.98, 2.88
M (OCplx, W_M=1)	7	[46-67]	13	[2.98-3.88]
	-	61, 62.8	-	3.61, 3.7
Ocplx, W_A= W_M=0.5	7	39	13	2.68
	-	-	-	-

Table 3. Summary of Results for the ATM.

3.3.2. The Ant System

We followed the exact same procedure to experiment with the Ant system, which is more complex. Results are summarized in Table 4. This table has the same structure as Table 3 but one important difference is that, due to the increase in complexity, we do not consistently obtain the best orders for any of the cost functions. Each cell therefore contains the interval of values we obtain across 100 GA executions. Average and median values are also provided below the intervals.

Results show that values for *D* are always close to the best-found value (10, the minimum determined by the GA and the graph-based algorithm), regardless of the cost function selected. The results in terms of number of attributes (*A*) are obviously very good when *A* is the cost function but also when a weighted function of *A* and *M* (*OCplx* with W_A=W_M=0.5) is used (this is highlighted in Table 4 with dark gray cells). However these two functions perform poorly in terms of methods (*M*). With respect to *M*, results are best when using *M* or *D* as cost functions (see light gray cells in Table 4), and these two functions produce poor results in terms of attributes (*A*) and the weighted stubbing complexity function (*OCplx*).

		Stubbing Complexity Values			
		D	A	M	OCplx
Cost Functions	D	[10-13] 11, 10.98	[152-274] 187, 192	[19-32] 26, 24.68	[3.97-6.42] 4.63, 4.69
	A (OCplx, W_A=1)	[12-14] 12, 12.27	[131-137] 131, 132.7	[29-37] 33, 32.7	[3.59-3.95] 3.74, 3.74
	M (OCplx, W_M=1)	[10-14] 13, 12.5	[163-235] 197, 204.4	[19-26] 22, 21.67	[4.1-5.53] 4.66, 4.83
	Ocplx, W_A= W_M=0.5	[12-14] 12, 12.16	[131-143] 136, 136.25	[29-35] 29, 29.45	[3.59-3.86] 3.59, 3.62

Table 4. Summary of Results for the Ant.

What we can conclude is that the results when using *OCplx* as a cost function seem to be mostly driven by *A*. Though using *OCplx* leads to very good results in terms of *D* and *A*, they are also significantly different from the best-found value with respect to *M*. After investigating the reasons in more depth, we have come to realize that orders that minimize *M* (in the [19-26] range) tend to break large numbers of relationships that are associated with the maximum number of attributes and therefore dramatically increase *A*. Recall from Figure 2 that the attribute coupling distribution for Ant showed a cluster of 12 extremely large values, far above most of the other relationships. No such pattern was observed for method coupling or attribute coupling distributions of the ATM. We can therefore understand that any order breaking such relationships is unlikely to be optimal when using *OCplx* as a cost function, even though we normalize and account for *A* and *M* with the same weight. In other words, due to the distributions of attribute coupling in Ant, *OCplx* is strongly driven by *A*, and optimizing *M* systematically leads to significantly poorer results with respect to *A*. Using *OCplx* yields good results with respect to *D* and *A*, but leads to a [29-35] range (with a 29 median) for *M*, instead of the [19-26] range (with a 22 median) we obtain using *M* as a cost function. Whether this is an acceptable compromise among the multiple objectives of our optimization (D, A, M) is a subjective call. What we can say objectively is that using *OCplx* (W_A=W_M=0.5) is still a compromise. This is not visible when looking at the intervals but is very clear when looking at the distributions of *M* values in Figure 3. Figures (a), (b) and (c) show the distributions when *M*, *A*, and *OCplx* are used as a cost function, respectively. The latter is clearly a compromise between the first two cases.

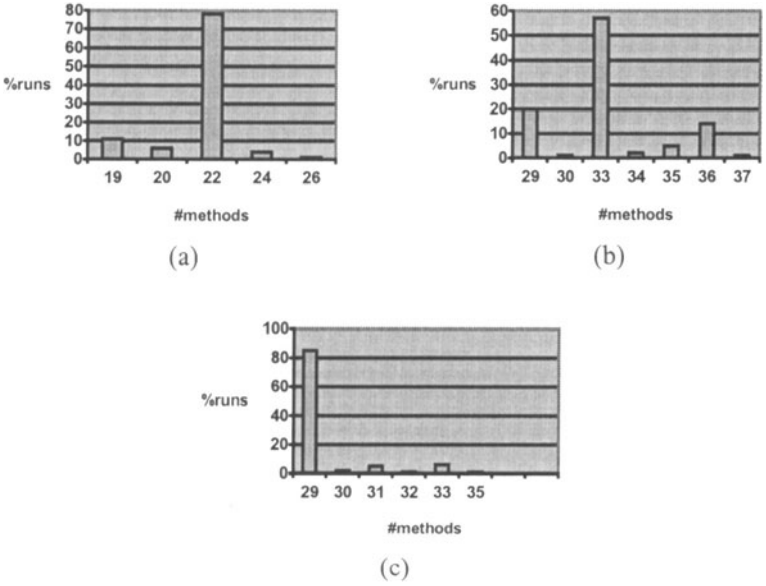


Figure 3. M Distributions when the cost function is M, A and Ocplx.

Another interesting point is that, because Ant is much more complex with respect to dependencies and cycles than the ATM system, we do not systematically obtain the best results across the 100 GA executions. However, we see that for best-found results (e.g., A∈[131-143] where the cost function is *OCplx*), intervals are also narrow. This implies that the variations observed in the outputs of the GA executions do not hamper its practical use as the orders obtained differ little in terms of stubbing complexity. Furthermore, we have observed that if the GA is executed a small number of times (say 10), we are very likely to obtain the minimal bound of the interval at least once. This is illustrated by Figure 4 where the distribution of *OCplx* values is shown for the 100 executions of the GA with *OCplx* as a cost function. We see that though the interval is [3.59-3.93], the minimal bound is very likely to occur.

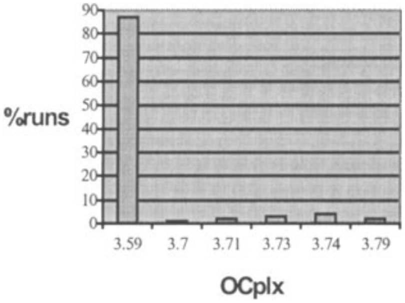


Figure 4. Distribution of OCplx across 100 GA executions for the Ant system.

When results significantly differ from the best-found values (e.g., A∈[163-235] when the cost function is *M*), intervals tend to be significantly larger as well, thus creating more uncertainty in the results the GA may produce.

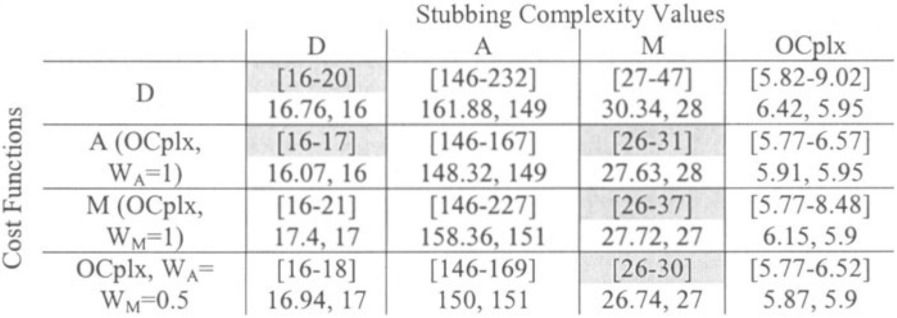
To conclude, we see that the results for Ant are not as clear-cut as for ATM. This is to be expected due to the higher complexity of Ant and is also explained by the specific pattern of dependencies within Ant that makes it difficult to fully optimize *M* and *A* within the same order. Despite those differences, GA's still appear to be useful to achieve compromises between *D*, *A*, and *M*.

3.3.3. The SPM System

Like in the Ant system, we do not consistently obtain best-found orders for any of the cost functions. As shown in Table 5, each cell therefore contains the interval of values we obtain across 100 GA executions. Average and median values are also provided below the intervals.

Results show that test orders with any one of *D, A, M* and *OCplx* as the cost function bring all evaluation criteria near the best-found values. This is just a coincidence as the attribute coupling matrix and the method coupling matrix are independent of each other.

It may seem strange that test orders with *D* as the cost function are even worse than those with *A* as the cost function when the number of broken dependencies is evaluated. It may also seem strange that test orders with *M* as the cost function are slightly worse than those with *OCplx* when the number of stubbed methods is evaluated (this is highlighted in gray cells in Table 5). Deriving test orders optimizing *A* happens to exclude a small number of test orders which are sub-optimal for *D* but which are sometimes produced when using *D* as the cost function. Likewise, deriving test orders optimizing *OCplx* happens to exclude a small number of test orders which are sub-optimal for *M*.



Cost Functions	Stubbing Complexity Values			
	D	A	M	OCplx
D	[16-20] 16.76, 16	[146-232] 161.88, 149	[27-47] 30.34, 28	[5.82-9.02] 6.42, 5.95
A (OCplx, W_A=1)	[16-17] 16.07, 16	[146-167] 148.32, 149	[26-31] 27.63, 28	[5.77-6.57] 5.91, 5.95
M (OCplx, W_M=1)	[16-21] 17.4, 17	[146-227] 158.36, 151	[26-37] 27.72, 27	[5.77-8.48] 6.15, 5.9
OCplx, W_A= W_M=0.5	[16-18] 16.94, 17	[146-169] 150, 151	[26-30] 26.74, 27	[5.77-6.52] 5.87, 5.9

Table 5. Summary of Results for the SPM.

3.3.4. The BCEL System

Once again, with the BCEL system, we do not consistently obtain the best-found orders for any of the cost functions. Each cell therefore contains the interval of values we obtain across 100 GA executions. Average and median values are also provided below the intervals (Table 6).

Results show that GA generated orders are close to the best-found orders for both *D* and *M* when *D* or *M* is used as the cost function. However, the results also show that GA generated orders for both *D* and *M* are comparatively poor when *A* is used as the cost function. It seems that orders that minimize the number of broken dependencies are likely to minimize the number of stubbed methods and vice versa. However, they yield significantly more attributes to be stubbed than those using *A* as the cost function. Furthermore, orders that minimize the number of stubbed attributes are not optimal with respect to *D* and *M*. This is also the reason why the orders determined by the *A*, *D* and *M* cost functions cannot minimize *OCplx*. Unlike the Ant example, the results when using *OCplx* as a cost function do not seem to be driven by any single coupling measure. But even then, using *OCplx* leads to reasonably good results in terms of every other coupling measure. Like in the Ant example, after investigating the attribute coupling and method coupling distributions (Figure 2), we find that there are more relationship clusters with high attribute coupling than those with high method coupling. This is why breaking dependencies with *M* as the cost function highly increases the number of stubbed attributes.

		Stubbing Complexity Values			
		D	A	M	OCplx
Cost Functions	D	[63-70] 65.5, 65.5	[101-143] 126.7, 127	[70-87] 76.5, 76	[8.59-10.28] 9.2, 9.1
	A (Ocplx, WA=1)	[71-82] 75.1, 75	[46-55] 49.0, 49	[76-105] 89.6, 89.5	[8.06-10.99] 9.4, 9.4
	M (Ocplx, WM=1)	[63-72] 67.8, 68	[49-144] 120.8, 127	[67-84] 74.4, 75	[8.08-10.14] 8.9, 8.9
	OCplx, WA= WM=0.5	[69-77] 72.1, 72	[46-96] 56.5, 53	[70-84] 78, 77	[7.56-9.08] 8.3, 8.2

Table 6. Summary of Results for the BCEL.

The intervals for this example are much wider than those found in the Ant example for each cost function (e.g., A ∊ [46- 96], *OCplx* ∊ [7.56-9.08], and the cost function is *OCplx*). Once again, the high complexity of the system (size, the number of dependency cycles), as described in Table 1, introduces more uncertainty in the GA results.

After investigation of the *OCplx* distribution when *OCplx* is used as a cost function (see Figure 5), we found that there is a 15% chance that *OCplx* will lie within 5% of the best-found result: [7.56-7.94]. Unlike in the Ant system, a larger number of executions (say, 20) is required to achieve a high likelihood of obtaining orders near best-found values. Since it takes, for the BCEL system, less than 20 minutes for each execution of the GA tool on a typical personal computer (500MHz, 128 MB), such a number of executions is still acceptable. Recall this system is the most complex, both in terms of size and number of dependency cycles. It is also expected that systems of such complexity are commonplace in the software industry.

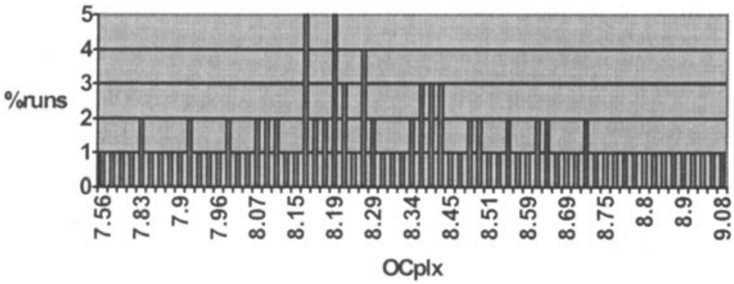


Figure 5. Distribution of *OCplx* for the BCEL system.

3.3.5. The DNS System

There are more classes in the DNS system than in any of the other systems under study. However, this system has the lowest number of cycles among our five examples. Consequently, result distributions tend to be narrow, like in the ATM example. Results are summarized in Table 7. Notice that the best-found result is consistently obtained for each cost function (no interval). When *D* is used as the cost function, the number of dependencies broken is the best (6). However, cost function D yields an increased number of stubbed attributes [19-28], while *OCplx* also increases as it lies within [1.47-1.99]. The suboptimal results for *A* may be explained by the fact that the attribute coupling distribution shows more variance than the method coupling distribution. Therefore, when using the number of dependencies broken as a cost function, we frequently obtain orders that are suboptimal in terms of stubbed attributes. As we see next, this can be addressed by accounting for both *A* and *M* in the cost function, which leads to orders that are consistently near the best-found value. The results in terms of number of stubbed methods always remain at the best-found value, regardless of the selected cost function. By examining the method coupling distribution (Figure 2), we find that the result is always the best because method coupling values are similar for most relationships. For instance, there are 113 out of 234 (see Table 1) breakable relationships with method coupling = 1, and 56 out of 234 breakable relationships with method coupling = 0. When the cost function is *M*, we consistently obtain the best-found result for *M* (11) as well as for *A* and *OCplx*. When the cost function is *A*, we consistently obtain the best-found result for *A* (39) as well as for *M*, but not for *D* and *OCplx*. When using *OCplx* as the cost function, we obtain a result close to that of *A*. This suggests, once again, that *OCplx* may be a useful, practical method to achieve a reasonable compromise among multiple objectives, as formalized by the coupling measures.

		Stubbing Complexity Values			
		D	A	M	OCplx
Cost Functions	D	6 -	[19-28] 23.62, 22	11 -	[1.47-1.99] 1.73, 1.64
	A (OCplx, W_A=1)	[6-7] 6.88, 7	19 -	11 -	1.47 -
	M (OCplx, W_M=1)	[6-7] 6.9, 7	[19-28] 23.74, 22	11 -	[1.47-1.99] 1.74, 1.64
	OCplx, W_A= W_M=0.5	[6-7] 6.89, 7	[19-22] 19.03, 19	11 -	1.47 -

Table 7. Summary of Results for the DNS System.

3.4. Determining Weights in Ocplx

We have seen that a practical issue when using a multiple objectives cost function is to determine appropriate weights. Equal weights may be adequate but higher flexibility may be required in some cases. If the number of coupling measures is small it is always possible to perform an exhaustive search to determine optimal weights. If the number of coupling measures is too large to consider such an option, a practical procedure could be as follows:

1. Run the Genetic Algorithm as parameterized in this paper, using each coupling measure (objective) independently as a cost function, in order to identify their minimal value for a set of classes.
2. Assign an equal weight to all coupling measures and use the geometric average as the cost function.
3. Check whether, for each objective (coupling measure), the value of *OCplx* is *reasonably* close to the minimal value (what is reasonable is of course subjective).
4. If this is the case, keep the weights as they are. If a particular coupling measure shows a departure from the minimal value that is deemed too large, then increase its relative weight and rerun the Genetic Algorithm.
5. Repeating this procedure (from 3) will hopefully converge towards acceptable values for each coupling measure, thus leading to acceptable test order.

Another possibility is of course to assign weights based on some other rationale, that is something that is an estimation (e.g., based on expert opinion) of the relative cost of each measurement unit of each coupling measure. But it is difficult to envisage at this point how this could be achieved. The procedure described above can be automated and helps achieve a balanced compromise between all coupling measures, each of them representing an optimization objective.

4. CONCLUSION

We have shown here that, if we want to use more sophisticated criteria to optimize class integration orders, we cannot keep relying on graph-based algorithms as described in [6]. As an alternative, we proposed a solution based on Genetic Algorithms and coupling measurement that seems to address our needs. Coupling measurement helps us differentiate stubs of varying complexity and Genetic Algorithms allow us to minimize complex cost functions based on such complexity measurement.

In this paper, we select a specific type of Genetic Algorithms (with permutation encoding) so as to fit our application and we assess the results of our strategy on five real application systems of non-trivial size and complexity. Results are very encouraging as they consistently show that Genetic Algorithms can be used to obtain optimal results in a reliable and systematic manner and reach acceptable compromises among multiple objectives (i.e., coupling measures). They also provide a lot of flexibility in terms of allowing the user to optimize the integration order under constraints. This is important in practice as integration orders have an impact on development and unit testing orders and the latter may be constrained by available resources shared with other projects.

Results also show that, as dependencies and cycles become more complex, Genetic Algorithms tend to produce less consistent results from run to run. This is to be expected as they are based on heuristics and those heuristics are expected to be sensitive to the complexity of the ordering problem to solve. However, the good news is that the variation observed is small and, in practice, if the user runs the algorithm a small number of times, she is likely to obtain the lower bound at least once and can therefore use it as an optimal integration order. Since a run does not take more than a few minutes of execution on a typical personal computer (500 MHz), this does not pose any practical problem.

Another limitation to consider is the fact that case studies as the one we present here are performed without knowing the actual global minimum, as this cannot be derived with problems of such complexity. To alleviate the problem, we observe the minimum values when using each coupling measure in isolation as a cost function and look at the optimality and consistency of the results we obtain when running the Genetic Algorithm a large number of times with complex, multi-objectives cost functions. In light of the complexity of the case studies (as measured by dependencies and cycles), the minima we obtain seem realistic. Furthermore, with respect to the number of dependencies that are “broken”, we are able to compare our results to those of graph-based algorithms and show we consistently obtain (nearly) identical results.

It is worth noting that, though Genetic Algorithms perform as well as graph-based algorithms under similar conditions and facilitate the use of more complex cost functions (e.g., using coupling measurement), this does not make graph-based approaches obsolete. The execution of graph-based approaches takes the order of a second on a typical personal computer and, as a consequence, may turn out to be very useful if quick results are required and simple cost functions are used (e.g., *D*). The adequacy of a technique is context dependent and should be driven by criteria such as the size and complexity of the system, the cost function, the time constraints under which a solution must be obtained, or the need to account for external constraints to devise a test order.

Future work encompasses performing large-scale simulations in order to confirm the generality of the results we have obtained here with cases studies. Other more realistic fitness/evaluation function, as well as other constraints (e.g., organizational constraints such as availability of personnel or other development resources), can also be investigated.

ACKNOWLEDGEMENT

This work was partly supported by the NSERC-CSER consortium and Mitel Networks through a CRD grant. Lionel Briand and Yvan Labiche were further supported by NSERC operational grants. This work is part of a larger project on testing object-oriented systems with the UML (TOTEM: http://www.sce.carleton.ca/Squall/Totem/Totem.html).

NOTES

[1] Any small change in the order (e.g., swapping two classes) may have a large impact on the cost function.

[2] Linear problems describe cases where the cost function to minimize is linearly related to input parameters.

[3] We do not count inherited attributes (and methods) as this would lead to counting them several times when measuring the stubbing complexity of an order, as described in the *OCplx* formula below.

[4] We count here the number of *elementary* circuits [26], i.e., each class appears once and only once in each circuit.

[5] We made a conscious effort not to use libraries but application systems, in order to use case studies representative of the type of systems on which the techniques would typically be used and so as to avoid the peculiar class diagram topologies encountered in libraries (e.g., in [20], 4 of the 6 systems used are libraries and, for example, the Java Library shows 8000 cycles that are broken using 7 stubs).

[6] As obtained by the GA, which may not necessarily be the global minimum values. Unless otherwise specified, in the reminder of the text, we will use the term minimum with this restriction in mind.

[7] These minimum values are also determined from executing the Genetic Algorithms 100 times with *OCplx*, *A* and *M* as cost functions, respectively.

REFERENCES

[1] T. Back, "The Interaction of Mutation Rate, Selection, and Self-Adaptation Within a Genetic Algorithm," *Proc. Parallel Problem Solving from Nature (PPSN'92)*, Brussels (Belgium), pp. 85-94, September 28-30, 1992.

[2] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, 2nd Ed., 1990.

[3] R. V. Binder, *Testing Object-Oriented Systems - Models, Patterns, and Tools*, Addison-Wesley, 1999.

[4] L. Briand, J. Daly and J. Wuest, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 25 (1), pp. 91-121, 1999.

[5] L. Briand, J. Feng and Y. Labiche, "Experimenting with Genetic Algorithms to Devise Optimal Integration Test Orders," Carleton University, Technical Report SCE-02-03, March, 2002, http://www.sce.carleton.ca/Squall/Articles/TR_SCE-02-03.pdf, a short version appeared in the proceedings of SEKE 2002.

[6] L. Briand, Y. Labiche and Y. Wang, "Revisiting Strategies for Ordering Class Integration Testing in the Presence of Dependency Cycles," *Proc. 12th International Symposium on Software Reliability Engineering (ISSRE)*, Hong Kong, pp. 287-296, November 27-30, 2001.

[7] L. Briand, Y. Labiche and Y. Wang, "Revisiting Strategies for Ordering Class Integration Testing in the Presence of Dependency Cycles," Carleton University, Technical Report SCE-01-02, September, 2002, http://www.sce.carleton.ca/Squall/Articles/TR_SCE-01-02.pdf.

[8] L. Chambers, *Practical Handbook of Genetic Algorithms*, vol. 1, CRC Press, 1995.

[9] L. Davis, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, 1991.

[10] K. A. De Jong, *Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Ph.D. Dissertation, The University of Michigan, 1975

[11] N. Deo, *Graph Theory With Applications to Engineering and Computer Science*, 1974.

[12] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[13] D. E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison Wesley, 1989.

[14] B. F. Jones (Ed.), *Special Issue on Metaheuristic Algorithms in Software Engineering*, Information and Software Technology, vol. 43 (14), 2001.

[15] P. C. Jorgensen and C. Erickson, "Object-Oriented Integration Testing," *Communications of the ACM*, vol. 37 (9), pp. 30-38, 1994.

[16] D. Kung, J. Gao, P. Hsia, J. Lin and Y. Toyoshima, "Class Firewall, test order, and regression testing of object-oriented programs," *Journal of Object-Oriented Programming*, vol. 8 (2), pp. 51-65, 1995.

[17] D. Kung, J. Gao, P. Hsia, Y. Toyoshima and C. Chen, "On Regression Testing of Object-Oriented Programs," *Journal of Systems Software*, vol. 32 (1), pp. 21-40, 1996.

[18] Y. Labiche, P. Thévenod-Fosse, H. Waeselynck and M.-H. Durand, "Testing Levels for Object-Oriented Software," *Proc. 22nd IEEE International Conference on Software Engineering (ICSE)*, Limerick (Ireland), pp. 136-145, June, 2000.

[19] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys, *The Traveling Salesman Problem*, Wiley, 1985.

[20] V. Le Hanh, K. Akif, Y. Le Traon and J. M. Jezequel, "Selecting an efficient OO integration testing strategy: an experimental comparison of actual strategies," *Proc. 15th European Conference for Object-Oriented Programming (ECOOP)*, Budapest (Hungary), pp. 381-401, June, 2001.

[21] Y. Le Traon, T. Jéron, J.-M. Jézéquel and P. Morel, "Efficient Object-Oriented Integration and Regression Testing," *IEEE Transactions on Reliability*, vol. 49 (1), pp. 12-25, 2000.

[22] J. D. Mc Gregor and D. A. Sykes, *A Practical Guide to Testing Object-Oriented Software*, Addison-Wesley, 2001.

[23] Palisade, Evolver, the Genetic Algorithm Super Solver, 1998.

[24] K.-C. Tai and F. J. Daniels, "Interclass Test Order for Object-Oriented Software," *Journal of Object-Oriented Programming*, vol. 12 (4), pp. 18-25,35, 1999.

[25] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM Journal on Computing*, vol. 1 (2), pp. 146-160, 1972.

[26] R. Tarjan, "Enumeration of the Elementary Circuits of a Directed Graph," *SIAM Journal on Computing*, vol. 2 (3), pp. 211-216, 1973.

[27] D. M. Tate and A. E. Smith, "Expected Allele Coverage and the Role of Mutation in Genetic Algorithms," *Proc. Fifth International Conference on Genetic Algorithms*, pp. 31-37, 1993.

[28] S. R. Thangiah and K. E. Nygard, "School bus routing using genetic algorithms," *Proc. Applications of Artificial Intelligence SPIE Conference*, pp. 387-398, 1992

Automated Test Reduction Using an Info-Fuzzy Network

Mark Last[1] and Abraham Kandel[2]

*[1]Department of Information Systems Engineering
Ben-Gurion University of the Negev
Beer-Sheva 84105, Israel
mlast@bgumail.bgu.ac.il*

*[2]Department of Computer Science and Engineering
University of South Florida
4202 E. Fowler Avenue, ENB 118
Tampa, FL 33620, USA
kandel@csee.usf.edu*

ABSTRACT

In today's software industry, design of black-box test cases is a manual activity, based mostly on human expertise, while automation tools are dedicated to execution of pre-planned tests only. However, the manual process of selecting test cases can rarely be considered as satisfactory both in terms of associated costs and the quality of produced software. This paper presents an attempt to automate a common task in black-box testing, namely reducing the number of combinatorial tests. Our approach is based on automated identification of relationships between inputs and outputs of a data-driven application. The set of input variables relevant to each output is extracted from execution data by a novel data mining algorithm called the info-fuzzy network (IFN). The proposed method does not require the knowledge of either the tested code, or the system specification, except for the list of software inputs and outputs. In the paper, we report the results of applying the proposed approach to a typical business application program.

KEYWORDS

Black-box Testing, Test Reduction, Combinatorial Testing, Input-Output Analysis, Info-Fuzzy Network.

1. INTRODUCTION

Computer programs are apparently the most complex tools ever built by humans since the Stone Age. Like any other tool, whether it is a stone knife, a car, or a microchip, software is not guaranteed to work forever and under any conditions. Even worse than that, it is not 100% reliable: we never know *when* we are going to face the next failure in an executed program. A program *fails* when it does not do what it is required to do [1]. The purpose of testing a program is to discover faults that cause the system to fail. A test is considered successful if it reveals a problem; tests that do not expose any faults are useless, since they provide no indication about the program correctness [2]. The process of fault discovery is usually followed by some actions, which are aimed at preventing the future occurrence of the detected faults.

The ultimate goal of software testing is to test a program *completely*, i.e. to verify that the program works correctly and there are no undiscovered errors left. However, this goal is beyond our reach, since the program correctness can never be demonstrated through software testing [3]. The reasons for that include such over complex issues like the size of the input domain, the number of possible paths through the program, and wrong or incomplete specifications. In practice, the tester has to choose a limited number of test cases. According to [2], an ideal test case should have a reasonable probability of catching an error along with being non-redundant, effective, and of a proper complexity.

If the structure of the tested program itself is used to build a test case, this is called a *white-box* (or *open-box*) approach. White-box testing techniques include path, branch, and statement testing [2]. However, identifying every distinct path even in a small program may be a prohibitive task. Thus, in testing the functionality of a component, a sub-system, or a complete system, the *black-box* approach is much more common: the tested system is fed with inputs as a "black box" and then evaluated by its outputs. However, generating a set of representative test cases for black-box testing of software systems is a non-trivial task [2]. A traditional manual technique of black-box testing is based on identifying *equivalence classes* for program inputs and then choosing test cases at the class boundaries (ibid). Other common strategies include random testing [4] and statistical testing [5], which is based on the operational profile of a program. Due to explosive number of possible black-box tests for any non-trivial program, some techniques for semi-automatic prioritization and reduction of test cases are being developed. As shown below, existing methods are not suitable for completely automated testing of large-scale programs and systems, since they are based on manual analysis of the source code and/or execution data along with other forms of human intervention in the test selection process.

In this paper, we introduce a novel approach to combinatorial test reduction, which does not require any manual analysis of the source code. The proposed approach is based on automated identification of input-output relationships from execution data of the tested program. It is designed for testing applications with multiple inputs and outputs, such as APIs, form-based web applets, and decision support systems (DSS). Testability of such systems is especially low when the number of inputs is much larger than the number of outputs [5]. We use a data mining algorithm called the info-fuzzy network (IFN) to determine the set of input variables relevant to each output. The test reduction potential of the info-fuzzy method is demonstrated on a typical business application program previously used by us in [13] for evaluating an automated testing "oracle" based upon an artificial neural network.

Section 2 below discusses existing techniques for automated test case generation and reduction. The info-fuzzy method of input-output analysis is briefly described in Section 3. A prototype of an info-fuzzy test reduction system is presented in Section 4. The method is applied to a credit screening application in Section 5. Finally, Section 6 presents conclusions and directions for future research.

2. AUTOMATED TECHNIQUES FOR TEST CASE SELECTION

Black-box methods test the program response to sets of inputs *without* looking at the code. As indicated in [11], the main problem associated with the black-box approach is the rapid increase in the number of possible tests for systems with multiple inputs and outputs. When the number of tests becomes very large, there is a need of *input set reduction*. One of the most common test generation strategies, random testing [4], can lead to a limited number of test cases. However, random testing does not assume any knowledge of the tested system and its specifications and thus is insufficient for validating safety-critical or mission-critical software (ibid).

The test generation method of [6] is based on the assumption that the specifications are known and given as Boolean expressions. This methodology includes a family of strategies for automated generation of test cases that are substantially smaller than exhaustive test cases.

Another approach to test reduction is to filter out test cases that are not likely to expose any faults [9]. The effectiveness of every test case can be estimated by an artificial neural network, which is trained on a complete set of classified test cases. An "oracle" (usually, a human tester) classifies errors exposed by each test case in the training set. This is a promising approach to test reduction, though the generalization capabilities of a trained network in a large input space are questionable [9].

An algorithm for simplification of a given test case is presented in [12]. The algorithm is aimed at finding *a minimal test case*, where removing any additional input would cause the failure to disappear. Applying the algorithm of [12] to each test case that failed can improve the efficiency of the debugging process, but it has nothing to do with the problem of reducing the exponential number of combinatorial tests.

As shown in [11], input-output analysis can be utilized to reduce the number of black-box tests in a data-driven system. Such systems include embedded (real-time) systems, application program interfaces (API), and form-based web applications. The basic idea of input-output analysis is to reduce the number of combinatorial tests by focusing on those input combinations that affect at least one system output. Alternatively, if the total number of tests is limited, the same approach can lead to an increase in the fault detection capability of randomized test cases. According to [11], there are several ways to determine input-output relationships. Thus, a tester can analyze system specifications, perform structural analysis of the system's source code (if available), and observe the results of system execution. All these manual and semi-automatic techniques require a lot of human effort and, still, do not guarantee detection of all existing relationships [11].

In this paper, we extend the approach of [11] by proposing a new method for simplification and reduction of combinatorial test cases, which is based on *automated data-driven identification* of input-output relationships from *execution results*. This task is particularly important whenever input-output relationships cannot be derived directly from software model, since it is too complex, outdated, or completely missing as often happens with "legacy" systems. An information-theoretic data mining algorithm [14] determines the subset of input variables relevant to each system output and the corresponding equivalence classes, by automated analysis of execution data.

In comparison with existing methods of input-output analysis covered by [11], our method automates the manual activity of *execution-oriented analysis*. We assume that data mining methods can reveal more input-output relationships in less time than a human tester, though both approaches do not guarantee detection of *all* existing relationships. Like the manual analysis of execution data, our approach requires no information about the system specifications, except for the set of input / output variables and their acceptable values. The proposed automation of input-output analysis is expected to save a considerable amount of human effort in the test case design process.

3. INFO-FUZZY METHOD OF INPUT-OUTPUT ANALYSIS

Our approach to input-output analysis is based on the information-fuzzy network (IFN) methodology [15]. The info-fuzzy method produces a ranked list of inputs relevant to a given output as a result of inducing a classification model named *info-fuzzy network (IFN)* from execution data. This automated method of input-output analysis does not require the knowledge of either the tested code, or the system specification, except for a list of program inputs and outputs along with their respective data types. The underlying assumption of the proposed methodology is that a *stable version* of the tested software is available for training the network. Otherwise, the induced model may be based on faulty execution data resulting in a sub-optimal set of test cases. Thus, test reduction with automated input-output analysis is most appropriate for the black-box form of *regression testing*, which is one of the most important and extensive forms of testing [16].

An info-fuzzy network (see Figure 1) has a root node, a changeable number of hidden layers (one layer for each selected input), and a target (output) layer representing the possible output values. Each node in the output layer may be associated with a constant value (e.g., "Florida" or "10"), a range of values (e.g., [10.2, 11.6]), or an action performed by the program (e.g., printing a document). There is no limit as to the maximum number of output nodes in an info-fuzzy network. The network in Figure 1 has three output nodes denoted by numbers 1, 2, and 3.

Each hidden layer consists of nodes representing equivalence classes of the corresponding input variable. In case of a continuous variable, the algorithm determines automatically the equivalence classes as contiguous sub-ranges of input values. For multi-valued nominal variables, each value is considered an equivalence class unless specified otherwise by the user. In Figure 1, we have two hidden layers (No. 1 and No. 2).

The final (terminal) nodes of the network represent non-redundant test cases (conjunctions of input values that produce distinct outputs). The five terminal nodes of Figure 1 include (1,1), (1,2), 2, (3,1), and (3,2). Unlike decision-tree classification models (see [17] and [18]), the network has interconnections between terminal and target nodes, which represent expected program outputs for each test case. For example, the connection (1,1) → 2 in Figure 1 means that we expect the output value of 2 for a test case where both input variables are equal to 1. The connectionist nature of IFN resembles the structure of a multi-layer neural network (see [19]). Accordingly, we characterize our model as a *network* and not as a *tree*.

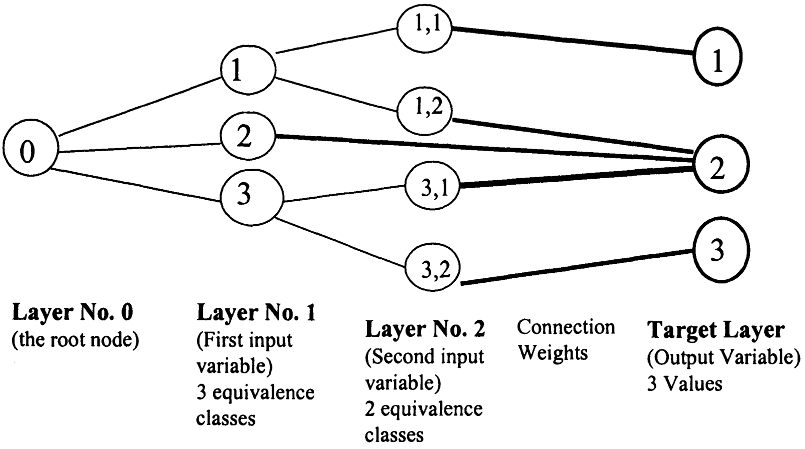


Figure 1. Info-Fuzzy Network - An Example.

The network is re-constructed completely for every output variable. The induction procedure starts with a single node representing an empty set of inputs. A node is split if it provides a statistically significant decrease in the conditional entropy [20] of the output. In information theory, the decrease in conditional entropy is called *conditional mutual information*. The IFN algorithm calculates the conditional mutual information of a candidate input attribute A_i and the output attribute T given a terminal node z by the following formula (based on [20]):

$$M\,(A_i;T/z)=\sum_{t=0}^{M_T-1}\sum_{j=0}^{M_i-1} P(C_t;V_{ij};z)\bullet\log\frac{P(V_{ij}^{t}/z)}{P(V_{ij}/z)\bullet P(C_t/z)} \tag{1}$$

where

M_T / M_i - number of distinct values of the output attribute T /candidate input attribute i.

$P\,(V_{ij}/z)$ - an estimated conditional (*a posteriori*) probability of a value j of the candidate input attribute i given the node z (also called a *relative frequency estimator*)

$P\,(C_t/z)$ - an estimated conditional (*a posteriori*) probability of a value t of the output attribute T given the node z.

$P\,(V_{ij}^{t}/z)$ - an estimated conditional (*a posteriori*) probability of a value j of the candidate input attribute i and a value t of the output attribute T given the node z.

$P\,(C_t\,;\,V_{ij}\,;\,z)$ - an estimated joint probability of a value t of the output attribute T, a value j of the candidate input attribute i, and the node z.

The statistical significance of the estimated conditional mutual information between a candidate input attribute A_i and the target attribute T, is evaluated by using the likelihood-ratio statistic (based on [21]):

$$G^2 (A_i ; T / z) = 2\bullet(\ln 2)\bullet E^*(z) \bullet MI (A_i ; T / z) \qquad (2)$$

where $E^*(z)$ is the number of cases associated with the node z.

The Likelihood-Ratio Test [22] is a general-purpose method for testing the null hypothesis H_0 that two random variables are statistically independent. If H_0 holds, then the likelihood-ratio test statistic $G^2 (A_i ; T / z)$ is distributed as chi-square with $(NI_i (z) - 1)\bullet(NT (z) - 1)$ degrees of freedom, where $NI_i (z)$ is the number of distinct values of a candidate input attribute i at node z and $NT (z)$ is the number of values of the target (output) attribute T at node z (see [24]). The default significance level (*p-value*), used by the IFN algorithm, is 0.1%.

A new input attribute is selected to maximize the total significant decrease in the conditional entropy, as a result of splitting the nodes of the last layer. The nodes of a new hidden layer are defined for a Cartesian product of split nodes of the previous hidden layer and values (equivalence classes) of the new input variable. If there is no candidate input variable significantly decreasing the conditional entropy of the output variable the network construction stops. In Figure 1, the first hidden layer has three nodes related to three possible values of the first input variable, but only nodes 1 and 3 are split, since the conditional mutual information as a result of splitting node 2 proves to be statistically insignificant. For each split node of the first layer, the algorithm has created two nodes in the second layer, which represent the two possible values of the second input variable. None of the four nodes of the second layer are split, because they do not provide a significant decrease in the conditional entropy of the output.

Our approach to automated determination of equivalence classes for numeric and other ordinal attributes is similar to the recursive discretization algorithm of [23]. IFN is looking for a partition of the input range that minimizes the conditional entropy of the output. The process of recursive partitioning is demonstrated in Figure 2 below: the best threshold is determined recursively for each sub-range of the input. Thus, the value T is chosen to split the entire attribute range into two intervals: below T and above T. The first interval is subsequently split into two sub-intervals, while the second interval is not. The stopping criterion used by IFN is different from [23]. As indicated above, we make use of a standard statistical *likelihood-ratio test* rather than searching for a *minimum description length*. The search for the best partition of a continuous attribute is *dynamic*: it is performed each time a candidate input attribute is considered for selection.

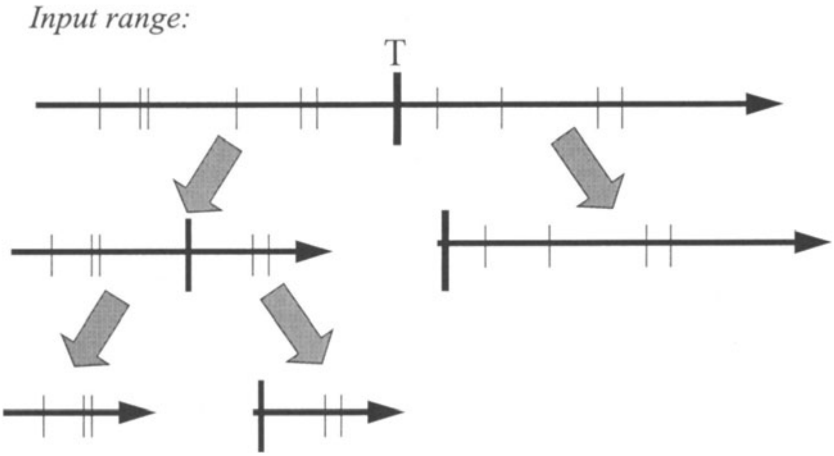


Figure 2. Recursive Discretization Algorithm.

The main steps of the network construction procedure are summarized in Table 1. Complete details are provided in [14] and [15].

Input:	The set of n execution runs; the set C of candidate inputs (discrete and continuous); the target (output) variable A_i; the minimum significance level sign for splitting a network node (default: sign = 0.1%).
Output:	A set I of selected inputs, equivalence classes for every input, and an info-fuzzy network. Each selected input has a corresponding hidden layer in the network.
Step 1	Initialize the info-fuzzy network (single root node representing all runs, no hidden layers, and a target layer for the values of the output variable). Initialize the set I of selected inputs as an empty set: $I = \varnothing$.
Step 2	While the number of layers $\|I\| < \|C\|$ (number of candidate inputs) do
Step 2.1	For each candidate input $A_{i'}$ /$A_{i'} \in C$; $A_{i'} \notin I$ do
	If $A_{i'}$ is continuous then Return the best equivalence classes of $A_{i'}$. Return statistically significant conditional mutual information cond_MI$_{i'}$ between $A_{i'}$ and the output A_i. End Do
Step 2.2	Find the candidate input $A_{i'}$* maximizing cond_MI$_{i'}$
Step 2.3	If cond_MI$_{i'*}$ = 0, then End Do. Else Expand the network by a new hidden layer associated with the input $A_{i'}$, and add $A_{i'}$ to the set I of selected inputs $I = I \cap A_{i'}$.
Step 2.4	End Do
Step 3	Return the set of selected inputs I, the associated equivalence classes, and the network structure

Table 1. Network Construction Algorithm

In [14], we have demonstrated the consistency and scalability of the IFN algorithm. Its run time is quadratic-polynomial in the number of inputs and linear in the number of outputs, which makes it appropriate for testing complex software systems with a large number of inputs and outputs. Other advantages of the IFN method include understandability and interpretability of results [26], stability of obtained models [27], and robustness to noisy and incomplete data [28]. In [29], the information-theoretic methodology is extended with a fuzzy-based technique for automated detection of unreliable output values. Consequently, IFN can be used as an automated "oracle" in black-box testing, but this application is beyond the scope of the work presented here.

4. REDUCING THE NUMBER OF TEST CASES

The info-fuzzy methodology of test reduction includes two parts (phases): the *training phase*, where we induce input-output relationships from execution data and the *evaluation phase* where we generate and run actual test cases based on the input-output analysis. Both parts are described in the sub-sections that follow.

4.1. Input-Output Analysis

The training phase of the IFN-based system for automated test case reduction is shown in Figure 3. Random Tests Generator (RTG) obtains the list of application inputs and their valid ranges from System Specification. No information about the expected system functionality is needed, since the IFN algorithm automatically reveals input-output relationships from randomly generated test cases. Finding the *minimal* number of random test cases required to cover all possible output values and execution paths of a given program with a sufficiently high probability is a subject of ongoing research. Systematic, non-random approaches to training set generation may also be considered.

The IFN algorithm is trained on inputs provided by RTG and outputs obtained from the tested application by means of the Test Execution module. As indicated above, a separate IFN model is built for each output variable. An IFN model includes a set of inputs relevant to the corresponding output, the associated equivalence classes determined by the algorithm, and the resulting set of non-redundant test cases.

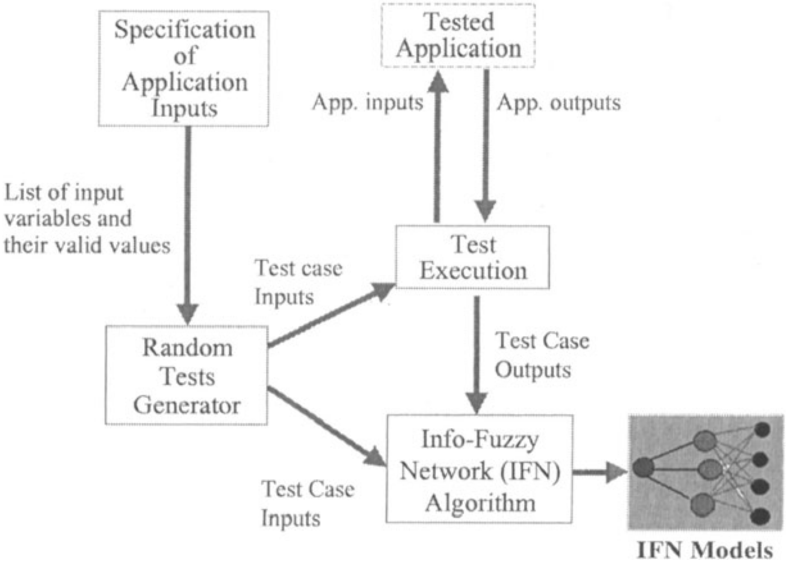


Figure 3. Test Reduction: Training Phase.

A brief description of each module in the system is provided below:

Specification of Application Inputs and Outputs (SAIO). Basic data on each input and output variable includes variable name, type (discrete, continuous, nominal, etc.), and a list or a range of possible values. Such information is generally available from requirements management and test management tools (e.g., Rational RequisitePro® or TestDirector®).

Random Tests Generator (RTG). This module generates random combinations of values in the range of each input variable. Variable ranges are obtained from the SAIO module (see above). The number of test cases to generate is determined by the user. The generated test cases are used by the Test Execution and the IFN modules.

Test Execution (TE). This module, sometimes called "test harness", feeds test cases generated by the RTG module to the tested application. The module obtains the application outputs for each test case and sends them to the IFN module.

Info-Fuzzy Network Algorithm (IFN). The input to the IFN algorithm includes the test cases randomly generated by the RTG module and the outputs produced by the tested application for each test case. IFN also uses the descriptions of variables stored by the SAIO module. The IFN algorithm is run repeatedly to find a subset of input variables relevant to each output and the corresponding set of non-redundant test cases. Actual test cases are generated from the automatically detected equivalence classes by using an existing testing policy (e.g., one test for each side of every equivalence class). A brief overview of the IFN methodology is provided in Section 3 above. Complete details can be found in [15].

4.2. Test Generation and Execution

Our approach is based on an assumption, which is true for many programs [11], that not all program inputs influence every output. Once the info-fuzzy algorithm has identified the input-output relationships, testing consists of generating test cases that represent the union of combinatorial tests for input variables included in the IFN model of each single output. The resulting test set is expected to be considerably smaller than the exhaustive test set. The evaluation phase of the IFN-based system for automated test reduction is shown in Figure 4. Test cases are generated by the Test Reducer (TR) module, which creates a union of tests based on all output variables. Like in the training phase, the Test Execution module feeds the tests cases to the tested application and reads the resulting outputs. The comparison between the expected and the actual outputs is performed by an "oracle", which can be either a human tester, or an automated system like the one presented by us in [13].

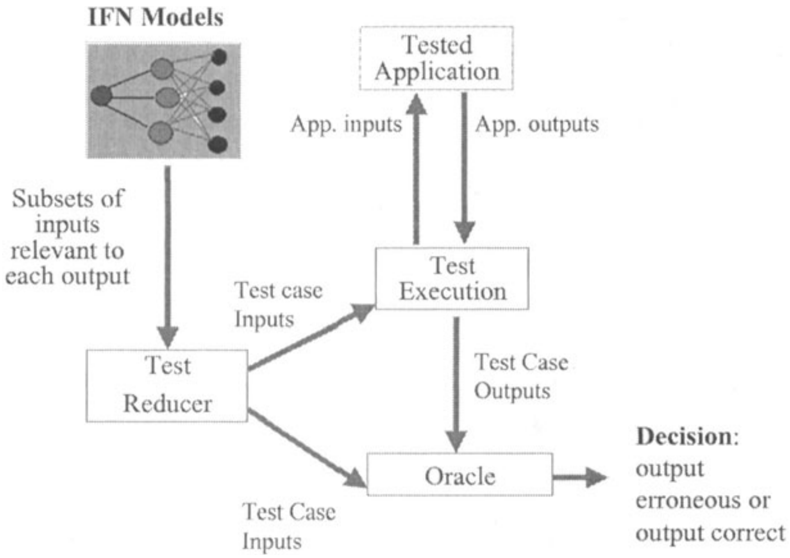


Figure 4. Test Reduction: Evaluation Phase.

A brief description of the Test Reducer module is given below:

Test Reducer (TR). This module creates a reduced set of combinatorial tests as a union of reduced tests for all output variables. The total number of reduced test cases to generate can be limited by the user. If the number of reduced combinatorial tests exceeds a user-specified limit, the module randomizes the values of individual input variables while restricting their combinations to the subsets of variables determined by the IFN module as influencing the application outputs. The generated test cases are submitted to the Test Execution module.

5. EXPERIMENTAL RESULTS

In this section, we present the results of using the info-fuzzy approach for automated test reduction in a typical business program. We start the section with describing the program used as our case study. Then we perform an automated input-output analysis by training the info-fuzzy algorithm. Finally, the induced info-fuzzy models are utilized for reducing the number of combinatorial tests. We evaluate the performance of the proposed method by the relative reduction in the amount of required combinatorial tests along with the fault detection potential of the reduced test set.

5.1. Description of Case Study

The program studied is called *Credit Approval*. Its task is to process a credit application of a potential credit card customer. The program has two output variables: *Decision* (approve / decline an application) and the amount of *Credit Limit* granted (greater than zero if an application is approved). We have previously used a similar program in [13] for evaluating the performance of an automated testing "oracle". This program is representative of a wide range of business applications, where a few critical outputs depend on a large number of inputs. In such applications, a reasonable assumption is that not every input is designed to affect every output. Moreover, due to continuous changes in user requirements, some inputs may become obsolete though they are still read and stored by the program.

The inputs of the studied application include eight attributes of the applicant such as age, income, citizenship, etc. Table 2 shows the complete list of variables used as input to the program, including their data type and range of possible values. Though this is a very small program (implemented with less than 300 lines of C code), it has as many as 52 distinct flow paths and the number of possible combinatorial tests exceeds 11 million (!). The core business logic of the program is shown as C code in Table 3.

No.	Name of Input Variable	Data Type	Total Number of Values	Details
1	Citizenship	Nominal	2	0: US 1: Other
2	State of Residence	Nominal	2	0: Florida 1: Other
3	Age	Continuous	100	1-100
4	Sex	Nominal	2	0: Female 1: Male
5	Region	Nominal	7	0-6 for different regions in the US
6	Annual Income	Continuous	200	$0k - $199k
7	Number of dependents	Continuous	5	0-4
8	Marital status	Nominal	2	0: Single 1: Married

Table 2. Descriptions of Input Variables.

```
if (region == 5 ||  region == 6)
 credlimit = 0;
else
  if (age < 18)
    credlimit = 0;
  else
     if (citizenship == 0)
     {
          credlimit = 5000 + 15*income;
          if (state == 0)
          {
              if (region == 3 || region ==4)
                {
                  credlimit = (int)(credlimit *
2.0);
                }
              else
                  credlimit = (int)(credlimit *
1.5);
          }
          else
              credlimit = (int)(credlimit * 1.1);
          if (marital_status == 0)
          {
          if (num_dep > 0)
                  credlimit = credlimit +
200*num_dep;
              else
              credlimit = credlimit + 500;
          }
          else
```

```
             credlimit = credlimit + 1000;
           if (sex == 0)
             credlimit = credlimit + 500;
           else
             credlimit = credlimit + 1000;
        }
        else
        {
          credlimit = 1000 + 12*income;
          if (marital_status == 0)
          {
            if (num_dep > 2)
              credlimit = credlimit +
100*num_dep;
            else
              credlimit = credlimit + 100;
          }
          else
            credlimit = credlimit + 300;
          if (sex == 0)
            credlimit = credlimit + 100;
          else
            credlimit = credlimit + 200;
        }
        if (credlimit == 0)
          decision = 1;
        else
          decision = 0;
```

Table 3. Core Business Logic.

5.2. Inducing Input-Output Relationships

To prepare a training set for the IFN algorithm (see Section 3 above), we have randomly generated 5,000 test cases in the input space of the Credit Approval program. As shown by the results below, this number was sufficient to identify all major execution paths in this program and to cover the corresponding output values. Determining the *minimal* number of random test cases required to perform input-output analysis of a given program is a part of our ongoing research. The info-fuzzy algorithm was run two times to find the inputs relevant to the two outputs of the program: *Decision* (Approve / Decline) and *Credit Limit*. Since IFN requires the output to be a discrete variable, we have discretized the continuous range of *Credit Limit* to 10 equally spaced sub-ranges of $2,000 width each ($1 - $2,000, $2,001 - $4,000, etc.). The large number of sub-ranges we have chosen is expected to be sufficient for detecting all inputs that have a significant impact on this particular output. As indicated above, IFN results are based solely on execution data and *not* on the actual code of the program shown in Table 3.

Only two inputs (*Region* and *Age*) were found to influence the first output (*Decision*). *Region* is a nominal attribute and, thus, the algorithm has referred to its each distinct value as an equivalence class. On the other hand, the range of *Age*, which can take 100 continuous values, has been partitioned into two equivalence classes only: customers below 18 years old vs. customers aged 18 years and older.

The info-fuzzy network built for the *Decision* output is shown in Figure 5. The 18 hidden nodes of the network include the root node (Node 0), nodes 1 –7 representing seven equivalence classes of the *Region* input, and nodes 8 – 17 standing for 10 combinations of five region classes with two equivalence classes of the *Age* input (0 – 17 vs. 18+). The algorithm has not split nodes 6 and 7, because in the two corresponding regions, the output does not depend on the applicant age. The network has 12 final (terminal) nodes numbered 6 – 17, which represent the 12 non-redundant test cases required for testing the correctness of *Decision* output. The minimal set of 12 non-redundant test cases is shown in Table 5 (see Appendix). The number of actual tests may be at least 22, since for each continuous equivalence class both boundary values should be tested [2]. The latter requirement implies that Nodes 8, 10, 12, 14, 16 are tested with *Age = 0* and *Age =17*, while Nodes 9, 11, 13, 15, 17 are tested with *Age = 18* and the oldest age possible (e.g., 100).

The target layer has two nodes: 0 (approve) and 1 (decline). Thin lines connecting final nodes to the target nodes show the expected program output for each corresponding test case. There is only one line exiting each final node, which means that the induced info-fuzzy network is a *perfect predictor* for the *Credit Approval* program with respect to the *Decision* output: the network output is always identical to the program output. This result is not true in a general case: for many applications, IFN may be unable to predict some of program outputs with 100% accuracy.

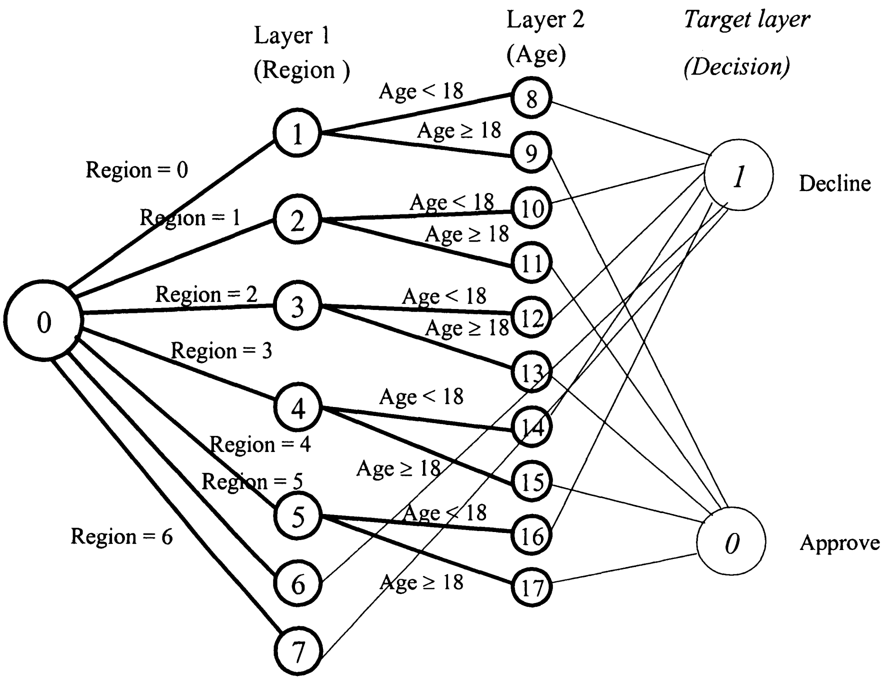


Figure 5. Info-Fuzzy Network (Output: Decision).

In the case of the second output (*Credit Limit*), the number of relevant inputs was found to be six out of eight candidates. Based on IFN results, the inputs affecting *Credit Limit* are: *Citizenship, Region, Age, Income, State,* and *Sex*. The algorithm has partitioned the ranges of two continuous variables (*Age* and *Income*) into two and ten equivalence classes respectively.

The info-fuzzy network induced for the *Credit Limit* output has six layers (associated with six selected inputs) and 239 hidden nodes, which include 165 final nodes. Due to its size, we cannot display the complete network here. The resulting set of 165 non-redundant test cases is shown in Table 6 (see Appendix). The testers may wish to increase the actual number of tests by testing each boundary value of every continuous equivalence class.

5.3. Reducing the Number of Test Cases

The final set of combinatorial tests is a union of tests for the two output variables (*Decision* and *Credit Limit*). As indicated in [11], the problem of finding the truly minimal test set is NP-hard. In our case study, one can easily verify that every test for *Decision* output (see Table 5) is included in the test set for *Credit Limit* (see Table 6). For example, tests 1 and 2 in Table 5 are covered by tests 1 and 2 in Table 6, test 12 in Table 5 is covered by test 60 in Table 6, etc. Thus, we can conclude that Table 6, with its 165 test cases, represents the union of the test sets in both tables. The resulting test set constitutes a reduction of 99.999% vs. the original number of 11 million combinatorial tests for this program!

We have evaluated the fault detection potential of the reduced test set by generating ten faulty versions of the original program. The list of faults injected in each version and the corresponding error rates (percentage of test cases in the test suite that have detected a fault) are shown in Table 4. The first line of Table 4 shows the error rate of the IFN model when applied to the outputs of the correct version. As mentioned above, the model is a perfect predictor of the first output (*Decision*). However, the second IFN output (*Credit Limit*) is wrong in 18.2% of test cases. This problem can be fixed manually by correcting the faulty expected values in the test suite before it is used for regression testing.

An injected fault is successfully detected by regression testing if the error rate of the test suite over the mutated version of the program is *greater* than the error rate over the original (correct) version. In practical terms, this means that there is at least one test case in the test suite that indicates the presence of a fault in the tested version. Since not every output is affected by a change in any line of code, the correctness of *all* program outputs should be evaluated in every test case. The results of Table 4 demonstrate that all injected faults have changed the value of the *Credit Limit* output in at least one test case, though the *Decision* output has been corrupted by the first four faults only. Thus in our experiments, we have gained a substantial reduction in the testing effort without dropping the fault detection capability of the full test set.

Fault #	**Line #**	**Original Line**	**Injected Fault**	**Fault Type**	**Error Rate (%)**	
					Output 1 (Decision)	**Output 2 (Credit Limit)**
			No faults		**0**	**0.182**
1	1	If ((Region == 5) || (Region == 6))	if(Region == 5)	Operator Change & Argument Change	0.059	0.186
2	1		if ((Region == 4) || (Region == 5))	Argument Change	0.118	0.349
3	4	If (Age < 18)	if(Age > 18)	Operator Change	0.588	0.624
4	4		if(Age < 25)	Argument Change	0.294	0.552
5	7	If (Citizenship == 0)	if(Citizenship == 1)	Argument Change	0	0.962
6	10	If (State == 0)	if (State == 1)	Argument Change	0	0.619
7	12	If ((Region == 3) || (Region == 4)	if (Region == 3)	Argument Change	0	0.233
8	12		if ((Region == 1) || (Region == 2))	Arguments Change	0	0.387
9	28	credlimit = credlimit + 500	credlimit = credlimit + 5000	Argument Change	0	0.401
10	34	credlimit = 1000 + 12*income	credlimit = 1000 + 2*income	Argument Change	0	0.377

Table 4. Summary of Experiments with Injected Faults.

6. CONCLUSIONS

In this paper, we have presented a novel approach to automated reduction of combinatorial black-box tests, based on automated identification of input-output relationships from execution data of the tested program. This approach is especially useful for regression testing of data-driven software systems with incomplete or missing specifications. We use a data mining algorithm called the info-fuzzy network (IFN) to determine the set of input variables relevant to each output. Additional benefits of the IFN algorithm include automated determination of continuous equivalence classes and selection of non-redundant test cases. The test reduction potential of the info-fuzzy method has been demonstrated on a typical business application program aimed at screening credit applicants.

In our future work, we plan to develop an automated test design system supporting the methodology presented in this paper. We also intend to enhance the info-fuzzy input-output analysis by automated determination of equivalence classes for nominal attributes. In addition, we plan to perform a set of extensive experiments with large software systems to study the impact of the proposed approach on the effectiveness and the productivity of the software testing process. Other research directions include reducing the size of the test set required for training the IFN algorithm and performing automated input-output analysis with other data mining techniques (e.g., artificial neural networks).

ACKNOWLEDGEMENT

This work was partially supported by the National Institute for Systems Test and Productivity at University of South Florida under the USA Space and Naval Warfare Systems Command Grant No. N00039-01-1-2248.

REFERENCES

[1] S. L. Pfleeger, “Software Engineering: Theory and Practice”, 2nd Ed., Prentice Hall, 2001

[2] C. Kaner, J. Falk, H.Q. Nguyen, “Testing Computer Software”, Wiley, 1999.

[3] R.A. DeMillo and A.J. Offlut, “Constraint-Based Automatic Test Data Generation”, IEEE Transactions on Software Engineering, Vol. 17, No. 9, pp. 900-910, 1991.

[4] E. Dustin, J. Rashka, J. Paul, “Automated Software Testing: Introduction, Management, and Performance”, Addison-Wesley, 1999.

[5] M. A. Friedman, J. M. Voas, “Software Assessment: Reliability, Safety, Testability”, Wiley, 1995.

[6] E. Weyuker, T. Goradia, and A. Singh, “Automatically Generating Test Data from a Boolean Specification”, IEEE Transactions on Software Engineering, Vol. 20, No. 5, pp. 353-363, 1994.

[7] J. M. Voas and G. McGraw, “Software Fault Injection: Inoculating Programs against Errors”, Wiley, 1998.

[8] S. Elbaum, A. G.Malishevsky, G. Rothermel, “Prioritizing Test Cases for Regression Testing”, Proc. of ISSTA '00, pp. 102-112, 2000.

[9] A. von Mayrhauser, C.W. Anderson, T. Chen, R. Mraz, C.A. Gideon, “On the Promise of Neural Networks to Support Software Testing”, In W. Pedrycz and J.F. Peters, editors, Computational Intelligence in Software Engineering, pp. 3-32, World Scientific, 1998.

[10] D. Hamlet, “What Can We Learn by Testing a Program?”, Proc. of ISSTA 98, pp. 50-52, 1998.

[11] P. J. Schroeder and B. Korel, “Black-Box Test Reduction Using Input-Output Analysis”, Proc. of ISSTA '00, pp. 173-177, 2000.

[12] R. Hildebrandt, A. Zeller, “Simplifying Failure-Inducing Input”, Proc. of ISSTA '00, pp. 135-145, 2000.

[13] M. Vanmali, M. Last, A. Kandel, “Using a Neural Network in the Software Testing Process”, International Journal of Intelligent Systems, Vol. 17, No. 1, pp. 45-62, 2002.

[14] M. Last, A. Kandel, O. Maimon, "Information-Theoretic Algorithm for Feature Selection", Pattern Recognition Letters, 22 (6-7), pp. 799-811, 2001.

[15] O. Maimon and M. Last, "Knowledge Discovery and Data Mining – The Info-Fuzzy Network (IFN) Methodology", Kluwer Academic Publishers, Massive Computing, Boston, December 2000.

[16] National Institute of Standards & Technology, "The Economic Impacts of Inadequate Infrastructure for Software Testing", Planning Report 02-3, May 2002.

[17] J.R. Quinlan, "Induction of Decision Trees", Machine Learning, vol. 1, no. 1, pp. 81-106, 1986.

[18] J. R. Quinlan, "C4.5: Programs for Machine Learning", Morgan Kaufmann, San Mateo, CA, 1993.

[19] T.M. Mitchell, "Machine Learning", McGraw-Hill, New York, 1997.

[20] T. M. Cover, "Elements of Information Theory", Wiley, New York, 1991.

[21] F. Attneave, "Applications of Information Theory to Psychology", Holt, Rinehart and Winston, 1959.

[22] C.R. Rao and H. Toutenburg, "Linear Models: Least Squares and Alternatives", Springer-Verlag, 1995.

[23] U. Fayyad and K. Irani, "Multi-Interval Discretization of Continuous-Valued Attributes for Classification Learning", Proc. of the 13th International Joint Conference on Artificial Intelligence, pp. 1022-1027, Morgan Kaufmann, 1993.

[24] W. Mendenhall, R.J. Beaver, B.M. Beaver, "Introduction to Probability and Statistics", Duxbury Press, 1999.

[25] M. Last and A. Kandel, "Fuzzification and Reduction of Information-Theoretic Rule Sets", in Data Mining and Computational Intelligence, A. Kandel, H. Bunke, and M. Last (Eds), Physica-Verlag, Studies in Fuzziness and Soft Computing, Vol. 68, pp. 63-93, 2001.

[26] M. Last, Y. Klein, A. Kandel, "Knowledge Discovery in Time Series Databases", IEEE Transactions on Systems, Man, and Cybernetics, Volume 31: Part B, No. 1, pp. 160-169, Feb. 2001.

[27] M. Last, O. Maimon, E. Minkov, "Improving Stability of Decision Trees", International Journal of Pattern Recognition and Artificial Intelligence, Vol. 16, No. 2, pp. 145-159, 2002.

[28] M. Last and A. Kandel, "Data Mining for Process and Quality Control in the Semiconductor Industry", In Data Mining for Design and Manufacturing: Methods and Applications, D. Braha (ed.), Kluwer Academic Publishers, pp. 207-234, 2001.

[29] M. Last and A. Kandel, "Automated Quality Assurance of Continuous Data", to appear in NATO Science Series book "Systematic Organization of Information in Fuzzy Systems".

APPENDIX

Case ID	Region	Age	Final Node	Expected Output
1	5	Any	6	1
2	6	Any	7	1
3	0	0	8	1
4	0	18	9	0
5	1	0	10	1
6	1	18	11	0
7	2	0	12	1
8	2	18	13	0
9	3	0	14	1
10	3	18	15	0
11	4	0	16	1
12	4	18	17	0

Table 5. Non-redundant test cases for Decision Output.

Case ID	Citizenship	Region	Age	Income	State	Sex	Final Node	Expected Output[1]
1	0	5	Any	Any	Any	Any	8	0
2	0	6	Any	Any	Any	Any	9	0
3	1	5	Any	Any	Any	Any	15	0
4	1	6	Any	Any	Any	Any	16	0
5	0	0	0	Any	Any	Any	17	0
6	0	1	0	Any	Any	Any	19	0
7	0	2	0	Any	Any	Any	21	0
8	0	3	0	Any	Any	Any	23	0
9	0	4	0	Any	Any	Any	25	0
10	1	0	0	Any	Any	Any	27	0
11	1	1	0	Any	Any	Any	29	0
12	1	2	0	Any	Any	Any	31	0
13	1	3	0	Any	Any	Any	33	0
14	1	4	0	Any	Any	Any	35	0
15	0	0	18	59	Any	Any	40	10001
16	0	0	18	147	Any	Any	44	8001
17	0	1	18	31	Any	Any	48	8001
18	0	1	18	147	Any	Any	54	8001
19	0	4	18	31	Any	Any	78	12001
20	1	0	18	0	Any	Any	87	1
21	1	0	18	31	Any	Any	88	1
22	1	0	18	42	Any	Any	89	2001
23	1	0	18	59	Any	Any	90	2001
24	1	0	18	67	Any	Any	91	2001

[1] Lower boundary of the corresponding interval

Case ID	Citizenship	Region	Age	Income	State	Sex	Final Node	Expected Output[1]
25	1	0	18	84	Any	Any	92	2001
26	1	0	18	121	Any	Any	93	2001
27	1	0	18	147	Any	Any	94	2001
28	1	0	18	154	Any	Any	95	2001
29	1	0	18	183	Any	Any	96	2001
30	1	1	18	0	Any	Any	97	1
31	1	1	18	31	Any	Any	98	1
32	1	1	18	42	Any	Any	99	1
33	1	1	18	59	Any	Any	100	2001
34	1	1	18	67	Any	Any	101	2001
35	1	1	18	84	Any	Any	102	2001
36	1	1	18	121	Any	Any	103	2001
37	1	1	18	147	Any	Any	104	2001
38	1	1	18	154	Any	Any	105	2001
39	1	1	18	183	Any	Any	106	2001
40	1	2	18	0	Any	Any	107	1
41	1	2	18	31	Any	Any	108	1
42	1	2	18	42	Any	Any	109	1
43	1	2	18	59	Any	Any	110	2001
44	1	2	18	67	Any	Any	111	2001
45	1	2	18	84	Any	Any	112	2001
46	1	2	18	121	Any	Any	113	2001
47	1	2	18	147	Any	Any	114	2001
48	1	2	18	154	Any	Any	115	2001
49	1	2	18	183	Any	Any	116	2001
50	1	3	18	0	Any	Any	117	1
51	1	3	18	31	Any	Any	118	1
52	1	3	18	42	Any	Any	119	1
53	1	3	18	59	Any	Any	120	2001
54	1	3	18	67	Any	Any	121	2001
55	1	3	18	84	Any	Any	122	2001
56	1	3	18	121	Any	Any	123	2001
57	1	3	18	147	Any	Any	124	2001
58	1	3	18	154	Any	Any	125	2001
59	1	3	18	183	Any	Any	126	2001
60	1	4	18	0	Any	Any	127	1
61	1	4	18	31	Any	Any	128	1
62	1	4	18	42	Any	Any	129	2001
63	1	4	18	59	Any	Any	130	2001
64	1	4	18	67	Any	Any	131	2001
65	1	4	18	84	Any	Any	132	2001
66	1	4	18	121	Any	Any	133	2001
67	1	4	18	147	Any	Any	134	2001
68	1	4	18	154	Any	Any	135	2001
69	1	4	18	183	Any	Any	136	2001
70	0	0	18	0	0	Any	137	8001
71	0	0	18	0	1	Any	138	6001
72	0	0	18	31	0	Any	139	8001
73	0	0	18	31	1	Any	140	6001

Case ID	Citizenship	Region	Age	Income	State	Sex	Final Node	Expected Output[1]
74	0	0	18	42	0	Any	141	10001
75	0	0	18	42	1	Any	142	6001
76	0	0	18	67	0	Any	143	10001
77	0	0	18	67	1	Any	144	8001
78	0	0	18	84	0	Any	145	10001
79	0	0	18	84	1	Any	146	8001
80	0	0	18	121	0	Any	147	10001
81	0	0	18	121	1	Any	148	8001
82	0	0	18	154	0	Any	149	12001
83	0	0	18	183	0	Any	151	12001
84	0	0	18	183	1	Any	152	10001
85	0	1	18	0	0	Any	153	8001
86	0	1	18	0	1	Any	154	6001
87	0	1	18	42	0	Any	155	10001
88	0	1	18	42	1	Any	156	6001
89	0	1	18	59	0	Any	157	10001
90	0	1	18	59	1	Any	158	6001
91	0	1	18	67	0	Any	159	10001
92	0	1	18	67	1	Any	160	8001
93	0	1	18	84	0	Any	161	10001
94	0	1	18	84	1	Any	162	8001
95	0	1	18	121	0	Any	163	12001
96	0	1	18	121	1	Any	164	8001
97	0	1	18	154	0	Any	165	12001
98	0	1	18	154	1	Any	166	8001
99	0	1	18	183	0	Any	167	12001
100	0	1	18	183	1	Any	168	10001
101	0	2	18	0	0	Any	169	8001
102	0	2	18	0	1	Any	170	6001
103	0	2	18	31	0	Any	171	8001
104	0	2	18	31	1	Any	172	6001
105	0	2	18	42	1	Any	174	6001
106	0	2	18	59	0	Any	175	10001
107	0	2	18	59	1	Any	176	6001
108	0	2	18	67	0	Any	177	10001
109	0	2	18	67	1	Any	178	8001
110	0	2	18	84	0	Any	179	10001
111	0	2	18	84	1	Any	180	8001
112	0	2	18	121	0	Any	181	12001
113	0	2	18	121	1	Any	182	8001
114	0	2	18	147	0	Any	183	12001
115	0	2	18	147	1	Any	184	8001
116	0	2	18	154	0	Any	185	12001
117	0	2	18	183	0	Any	187	12001
118	0	2	18	183	1	Any	188	10001
119	0	3	18	0	0	Any	189	10001
120	0	3	18	0	1	Any	190	6001
121	0	3	18	31	0	Any	191	12001
122	0	3	18	31	1	Any	192	6001

Case ID	Citizenship	Region	Age	Income	State	Sex	Final Node	Expected Output[1]
123	0	3	18	42	0	Any	193	12001
124	0	3	18	42	1	Any	194	6001
125	0	3	18	59	0	Any	195	12001
126	0	3	18	59	1	Any	196	8001
127	0	3	18	67	0	Any	197	12001
128	0	3	18	67	1	Any	198	8001
129	0	3	18	84	0	Any	199	14001
130	0	3	18	84	1	Any	200	8001
131	0	3	18	121	0	Any	201	14001
132	0	3	18	121	1	Any	202	8001
133	0	3	18	147	1	Any	204	8001
134	0	3	18	154	0	Any	205	16001
135	0	3	18	183	0	Any	207	16001
136	0	3	18	183	1	Any	208	10001
137	0	4	18	0	0	Any	209	10001
138	0	4	18	0	1	Any	210	6001
139	0	4	18	42	0	Any	211	12001
140	0	4	18	42	1	Any	212	6001
141	0	4	18	59	0	Any	213	12001
142	0	4	18	59	1	Any	214	6001
143	0	4	18	67	0	Any	215	12001
144	0	4	18	67	1	Any	216	8001
145	0	4	18	84	0	Any	217	14001
146	0	4	18	84	1	Any	218	8001
147	0	4	18	121	0	Any	219	14001
148	0	4	18	121	1	Any	220	8001
149	0	4	18	147	0	Any	221	14001
150	0	4	18	147	1	Any	222	8001
151	0	4	18	154	0	Any	223	16001
152	0	4	18	183	0	Any	225	16001
153	0	4	18	183	1	Any	226	10001
154	0	0	18	154	1	0	227	8001
155	0	0	18	154	1	1	228	10001
156	0	2	18	42	0	0	229	8001
157	0	2	18	42	0	1	230	10001
158	0	2	18	154	1	0	231	8001
159	0	2	18	154	1	1	232	10001
160	0	3	18	147	0	0	233	14001
161	0	3	18	147	0	1	234	16001
162	0	3	18	154	1	0	235	8001
163	0	3	18	154	1	1	236	10001
164	0	4	18	154	1	0	237	8001
165	0	4	18	154	1	1	238	10001

Table 6. Non-redundant test cases for Credit Limit Output.

A Genetic Algorithm Approach to Focused Software Usage Testing

Robert M. Patton, Annie S. Wu, and Gwendolyn H. Walton

University of Central Florida
School of Electrical Engineering and Computer Science
Orlando, FL, U.S.A
rmpatton@yahoo.com

ABSTRACT

Because software system testing typically consists of only a very small sample from the set of possible scenarios of system use, it can be difficult or impossible to generalize the test results from a limited amount of testing based on high-level usage models. It can also be very difficult to determine the nature and location of the errors that caused any failures experienced during system testing (and therefore very difficult for the developers to find and fix these errors). To address these issues, this paper presents a Genetic Algorithm (GA) approach to focused software usage testing. Based on the results of macro-level software system testing, a GA is used to select additional test cases to focus on the behavior around the initial test cases to assist in identifying and characterizing the types of test cases that induce system failures (if any) and the types of test cases that do not induce system failures. Whether or not any failures are experienced, this GA approach supports increased test automation and provides increased evidence to support reasoning about the overall quality of the software. When failures are experienced, the approach can improve the efficiency of debugging activities by providing information about similar, but different, test cases that reveal faults in the software and about the input values that triggered the faults to induce failures.

KEYWORDS:

Genetic algorithms, software usage testing, simulation testing, debugging, system testing, black box testing

1. OVERVIEW

This work focuses on system level, model-based usage testing. The software to be tested is viewed from the perspective of the user as a black box system that operates in a specific environment, receives input, and provides output. One or more state-based models of software use are developed, using domain-specific knowledge to characterize the population of uses of the software (or usage scenarios) and to describe test management objectives and constraints. The usage models are used to assist with test planning, to generate a sample of test cases that represent usage scenarios, and to support reasoning about test results.

System-level usage testing approaches have proven to be successful for supporting test case selection and reasoning about test results in a variety of software projects. However, the system testing typically consists of only a very small sample from the set of possible scenarios of system use. Thus, it can be difficult or impossible to generalize the test results from a limited amount of testing based on high-level usage models. It can also be very difficult to determine the nature and location of the errors that caused any failures experienced during system testing (and therefore very difficult for the developers to find and fix these errors).

This paper presents a Genetic Algorithm (GA) approach to addresses these issues. As illustrated in Figure 1 and described in detail in section 5, the GA accepts input from two sources: (a) domain data generated by the usage model to define a usage scenarios and (b) the results (pass/fail) of system test. The initial population is defined as a set of test cases generated from a usage model. Each *individual* in the population represents a *single* test case. The individual is sent to the Tester to be processed and supplied to the Software Under Test. The Software Under Test processes this input and provides output that is analyzed for correctness by the Test Oracle. The Test Oracle will determine if the output is correct or flawed or if the software under test crashed. The Test Oracle informs the GA of the result: output is correct, output is flawed, or Software Under Test crashed. The GA uses this result along with the likelihood that it would occur as defined by the usage model to help determine the overall fitness of the individual. The GA outputs individual test cases that caused high intensity failures within the high usage areas of the software, thus driving dynamic testing and system analyses in a focused manner based on test objectives (as described by the usage model) and previous test results.

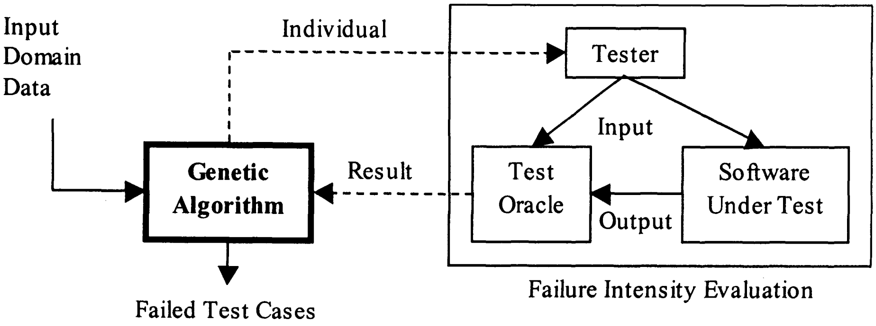


Figure 1. GA Approach to Focused Software Usage Testing.

The remainder of this paper is organized as follows. Section 2 provides a high-level introduction to Genetic Algorithms and pointers to related work. Section 3 provides some background information about system testing and debugging activities and challenges that motivate the GA approach presented in this paper. Section 4 introduces the GA approach to focused usage testing, and section 5 provides information about the internal details of the GA. Section 6 provides an example to illustrate application of this approach to drive focused testing of a military simulation system. Conclusions are presented in section 7.

2. INTRODUCTION TO GENETIC ALGORITHMS

A genetic algorithm (GA) is a search algorithm based on principles from natural selection and genetic reproduction [Holland 1975; Goldberg 1989]. GAs have been successfully applied to a wide range of applications, [Haupt 1998; Karr 1999; Chambers 2000] including optimization, scheduling, and design problems. Key features that distinguish GAs from other search methods include:

- A *population of individuals* where each individual represents a potential solution to the problem to be solved.
- A *fitness function* which evaluates the utility of each individual as a solution.
- A *selection function* which selects individuals for reproduction based on their fitness.
- Idealized *genetic operators* which alter selected individuals to create new individuals for further testing. These operators, e.g. crossover and mutation, attempt to explore the search space without completely losing information (partial solutions) that is already found.

Figure 2 provides the basic steps of a GA. First the population is initialized, either randomly or with user-defined individuals. The GA then iterates thru an evaluate-select-reproduce cycle until either a user defined stopping condition is satisfied or the maximum number of allowed generations is exceeded.

```
procedure GA
{
  initialize population;
  while termination condition not satisfied do
  {
         evaluate current population;
         select parents;
         apply genetic operators to parents to create children;
         set current population equal to be the new child population;
  }
}
```

Figure 2. Basic steps of a typical genetic algorithm.

The use of a population allows the GA to perform parallel searches into multiple regions of the solution space. Operators such as crossover [Holland 1975; Goldberg 1989; Mitchell 1996] allow the GA to combine discovered partial solutions into more complete solutions. As a result, the GA is expected to search for small building blocks in parallel, and then iteratively recombine small building blocks to form larger and larger building blocks. In the process, the GA attempts to maintain a balance between exploration for new information and exploitation of existing information. Over time, the GA is able to evolve populations containing more fit individuals or better solutions. For more information about GAs, the reader is referred to [Holland 1975; Goldberg 1989; Mitchell 1996; Coley 2001].

While, the GA approach presented in this paper is unlike other published approaches to the application of GA to support software testing or software quality assessment, the "failure-pursuit sampling" work of [Dickinson et al. 2001] and the "adaptive testing" work of [Schultz et al. 1992] are particularly noteworthy with respect to their motivation for the work of this paper.

While [Dickinson et al. 2001] does not explicitly make use of a GA, their concept of failure-pursuit sampling helped to provide a foundation for the approach presented in this paper. In failure-pursuit sampling, some initial sample of test cases is selected; the sample is evaluated and failures recorded; and additional samples are then selected that are in the vicinity of failures that occurred in the previous sample.

[Schultz et al. 1992] demonstrated the use of adaptive testing to test intelligent controllers for autonomous vehicles by creating individuals in the population that represented fault scenarios to be supplied to simulators of the autonomous vehicles. A benefit of such testing was to provide more information to the developers. According to [Schultz et al. 1992],

> "In more of a qualitative affirmation of the method, the original designer of the AUTOACE intelligent controller was shown some

of the interesting scenarios generated by the GA, and acknowledges that they gave insight into areas of the intelligent controller that could be improved. In particular, the scenarios as a group tend to indicate classes of weaknesses, as opposed to only highlighting single weaknesses. This allows the controller designers to improve the robustness of the controller over a class as opposed to only patching very specific instances of problems."

3. TESTING AND DEBUGGING CHALLENGES

Reasoning about the overall quality of a system can be difficult. For example, suppose a system accepts some data value X, and that the user profile for this system specifies that user is likely to use values in the range $30 < X < 70$. A usage model may generate two test cases that specify X as 40 and 60. If both of these test cases pass, it is not necessarily true that test cases will pass for all values of X. Similarly, if both of these test cases fail, it is not necessarily true that test cases will fail for all values of X. Additional focused testing (using similar, but different, test cases to identify more precisely the usage scenarios that induce failures and the scenarios that do not induce failures) may be necessary to support reasoning about the overall quality of the software.

In addition, in the situation when failures are observed during system testing, more testing can be required in order to precisely determine nature and location of the error(s) that caused the failures so the developers can find and fix the error. This find-and-fix process is referred to as "debugging". According to [Myers 1979], "of all the software-development activities, [debugging] is the most mentally taxing activity." This statement is often true today and can be the source of software quality problems. Test cases that reveal failures are often dissimilar to each other, the test results often provide little information concerning the cause of the failure and whether a similar scenario would fail in a similar manner. Without additional information, and with limited development resources, developers may be tempted to apply a small patch to the software to work around the failure rather than perform the analyses necessary to support complete understanding and correction of the problems that caused the failures.

A competitive mentality of "developers versus testers" often exists during testing. Because debugging requires additional information concerning the usage of the system and performing additional testing, once failures occur and the system must be corrected, this mentality should transition to "developers *and* testers versus the system" to facilitate the debugging effort. Developers often need the support of the testers during debugging because the developers may not have the necessary testing resources to do additional system level testing, or additional information concerning the usage of the system. As described by [Zeller 2001],

> "Testing is another way to gather knowledge about a program because it helps weed out the circumstances that aren't relevant to a particular failure. If testing reveals that only three of 25 user actions are relevant, for example, you can focus your search for the failure's root cause on the program parts associated with these three actions. If you can automate the search process, so much the better."

This description is consistent with the often-used induction approach to debugging described by [Myers 1979]. The induction approach begins by locating all relevant evidence concerning correct and incorrect system performance. As noted by [Myers 1979], "valuable clues are provided by similar, but different, test cases that *do not* cause the symptoms to appear. It is also useful to identify similar, but different, test case that *do* cause the symptoms to appear.

Similar to the notion of taking several "snapshots" of the evidence from different angles and with different magnification to look for clues from different perspectives, the debugging team needs to follow up on any failures identified during testing by more finely partitioning the input domain according to test results. This yields new evidence to be compared and organized in an attempt to identify and characterize patterns in the system's behavior. The next step is to develop a hypothesis about the cause of an observed failure by using the relationships among the observed evidence and patterns. Analyses can then be performed to prove that the hypothesis completely explains the observed evidence and patterns.

In practice, debugging can be very time-consuming, tedious, and error-prone when system-level testing reveals failures. Success of the debugging activity depends critically upon the first step in the process: the collection of evidence concerning correct and incorrect system performance. Assuming the total amount of evidence is manageable, an increase in useful evidence about correct and incorrect system performance can make it easier to identify patterns and develop and prove hypotheses. Thus, a mechanism is needed to drive testing and system analyses in a focused manner based on previous test results.

4. USING A GA FOR FOCUSED SOFTWARE USAGE TESTING

The genetic algorithm (GA) approach described in this paper drives dynamic generation of test cases by focusing the testing on high usage (frequency) and fault-prone (severity) areas of the software. This GA approach can be described as analogous to the application of a microscope. The microscope user first quickly examines an artifact at a macro-level to locate any potential problems. Then the user increases the magnification to isolate and characterize these problems.

Using the GA approach to focused software usage testing, the macro-level examination of the software system is performed using the organization's traditional model-based usage testing methods. Based on the results of this macro-level examination, a genetic algorithm is used to select additional test cases to *focus* on the behavior around the initial test cases to assist in identifying and characterizing the types of test cases that induce system failures (if any) and the types of test cases that do not induce system failures. If failures are identified, the genetic algorithm *increases the magnification* by selecting certain test cases for further analysis of failures. This supports isolation and characterization of any failure clusters that may exist.

Whether or not any failures are experienced, this genetic algorithm approach provides increased evidence for the testing team and managers to support reasoning about the overall quality of the software. In the situation where failures are experienced, the genetic algorithm approach yields information about similar, but different, test cases that reveal faults in the software and about the input values that triggered the faults to induce failures. This information can assist the developer in identifying patterns in the system's behavior and in devising and proving a hypothesis concerning the faults that caused the failures.

Because different software failures vary in severity to the user and in frequency of occurrence under certain usage profiles, certain failures can be more important than others. Factors such as the development team's uncertainty about particular requirements, complexity of particular sections of the code, and varying skills of the software development team can result in clusters of failure in certain partitions of the set of possible use of the software. As discussed in section 5.4 and section 5.5.1, the genetic algorithm's fitness function and selection function can address this issue, and help support the generation of test cases to identify failure clusters.

In the case of usage testing, highly fit individuals in the population are those that maximize two objectives. The first objective is likelihood of occurrence. Maximizing this objective means that the test case individual represents a scenario that closely resembles what the user will do with the system. The second objective is failure intensity (defined as a combination of failure density and failure severity). Maximizing this objective means that the test case individual has revealed spectacular failures in the system. Highly fit individuals with respect to the rest of the population are those that maximize both objectives as much as possible. To address this issue, a multi-objective GA technique [Fonseca 1995; Deb 1999; Coello Coello et al. 2002] is needed. As described in section 5.4, this application makes use of a nonlinear aggregating fitness combination [Coello Coello et al. 2002] to handle multiple objectives.

Furthermore, the purpose of the GA in this application is not to find a single dominant individual. This does not make sense from a testing perspective. Instead, the purpose is to locate and maintain a group of individuals that are highly fit. To do so, the GA for this application uses niching [Holland 1975; Horn 1994; Mahfoud 1995]. A niche represents some subpopulation of individuals who are similar, but different. As the GA runs, the most dominant niches (not the most dominant individual) survive. Niching used for this application is described in section 5.4.3.

The GA approach is applicable to testing many types of software. For example, in section 6 illustrative examples are presented of the application of a GA to support high-level usage testing of a military simulation system. For this case study, the test cases for a military simulation system consists of a variety of scenarios involving entities such as tanks, aircraft, armored personnel carries, and soldiers. Each entity can perform a variety of tasks. At a basic level, these scenarios involve some primary actor performing a task that may or may not involve a secondary actor, depending on the task. Each scenario is performed on a specific terrain map. For example, a scenario may consist of using a terrain map of Fort Knox with an M1A1 tank performing an Assault on a T-80 tank. In this example, the M1A1 tank is the primary actor since it performs the task (Assault), and it is the focus of the scenario. The T-80 tank is the secondary actor.

5. GA APPROACH DETAILS

To implement the GA for this case study example, a number of issues had to resolved, including the encoding of real world data, population initialization, fitness evaluation, and the use and operation of genetic operators. The following subsections discuss these issues and describe the internal details of the genetic algorithm.

5.1. Input Domain Data

As illustrated in Figure 3, there are three sources for the Input Domain Data that serves as input to the GA application shown in Figure 1. First, there is data that represents the bounds of the input domain for the software under test. This boundary data set does not necessarily specify all possible data values; rather it could merely specify the extreme values. For example, suppose the system accepts some data value *X*. Then the input boundary data might specify $0 < X < 10$. Second, there is data that represents the user's profile. This data defines what input data the user is likely to use and, implicitly, what data the user is not likely to use. For the previous example of the data value X, the user profile may specify $3 < X < 7$. The third source of input domain data is the set of test cases generated according to the user profile. For example, there may be two test cases that specify X as 4 and 6. The test cases and user profile data sets must be subsets of the input boundary data set.

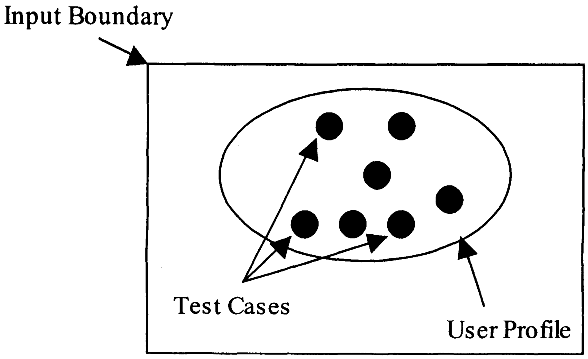


Figure 3. Input Domain Data.

Each of these three sources of input domain data is used for a specific purpose. The test cases are used to initialize the population. The user profile data set is used to help evaluate the fitness of individuals, specifically used to determine likelihood of occurrence. This causes the GA to focus its search to a particular area of the input domain. The input boundary data set is used to validate that new individuals are consistent with what the software under test allows the user to do. If an individual is created that lies outside of the defined input boundary data set, then that individual will be discarded by the GA.

5.2. Encoding

The test cases generated by the usage model are converted to an encoding based on real numbers for use in the GA population. This type of encoding was used so that there is a one to one correspondence between the gene and the variable it represents. In addition, it eliminates the problem of Hamming cliffs [Goldberg 1990]. Table 1, Table 2, and Table 3 illustrate a sample of the assigned identification numbers (IDs) for use in the GA.

Terrain ID Number	Terrain Map
1	NTC
2	Knox
3	Hunter
4	Itsec

Table 1. Terrain Identification Numbers.

Entity ID Number	Simulation Entity
2	M1A1
6	T-80
9	M3A3
17	SA-9
27	UH-60

Table 2. Entity Identification Numbers.

Task ID Number	Task
1	Move
3	Assault
7	Attack
11	Hover
15	Suppressive Fire

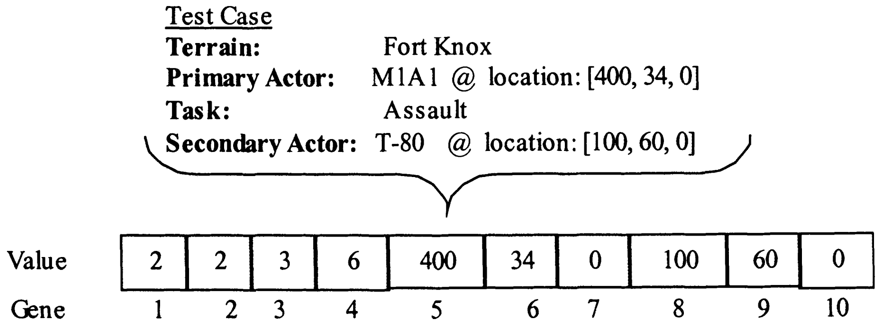
Table 3. Task Identification Numbers.

The individuals in the population of the GA consist of variations of these IDs. There are ten genes in each individual. The genotype is shown in Table 4.

Gene	Meaning	Valid Value Range
1	Terrain	Values 1 – 4
2	Primary Actor	Values 1 – 37
3	Task	Values 1 – 17
4	Secondary Actor	Values 0 – 37
5	X_1	Values greater than or equal to 0
6	Y_1	Values greater than or equal to 0
7	Z_1	Values greater than or equal to 0
8	X_2	Values greater than or equal to 0
9	Y_2	Values greater than or equal to 0
10	Z_2	Values greater than or equal to 0

Table 4. Genotype for individuals in the genetic algorithm.

Each gene represents an input value that a user could supply to the software being tested. The collection of ten genes represents a specific simulation scenario that may be run by the user on the Software Under Test. For example, Gene 1 represents the terrain map selected by the user. Gene 2 represents the primary actor selected by the user, such as a tank (i.e., M1A1, T-80), plane, helicopter, etc. Gene 3 represents the task assigned by the user to the primary actor, such as Move, Attack, Transport, etc. If the selected task requires a secondary actor, the user selects another actor, such as an enemy tank, enemy plane, friendly soldier, etc. Gene 4 represents the selected secondary actor. If the selected task does not require a secondary actor, Gene 4 is assigned a zero value. Genes 5 – 7 specify the location of the primary actor on the terrain map. If there is a secondary actor involved, then Genes 8 – 10 specify the location of the secondary actor on the terrain map. If there is no secondary actor, then Genes 8 – 10 represent some destination location that the primary actor must reach. An example of an individual is shown in Figure 4. This individual represents a scenario with an M1A1 tank assaulting a T-80 tank on the Fort Knox terrain map. The values shown in the first 4 genes of the individual are taken from Table 1, Table 2 and Table 3. The values for genes 5 – 10 are taken from the location values specified by the test case.



Test Case
Terrain: Fort Knox
Primary Actor: M1A1 @ location: [400, 34, 0]
Task: Assault
Secondary Actor: T-80 @ location: [100, 60, 0]

Value	2	2	3	6	400	34	0	100	60	0
Gene	1	2	3	4	5	6	7	8	9	10

Figure 4. Representation of test cases within the genetic algorithm.

Invalid individuals are discarded. For example, because a tank cannot attack an aircraft, an individual that represents this scenario would be discarded. Other invalid scenarios are those that specify locations (Genes 5 – 10) that lie outside the bounds of the terrain map. In addition, land vehicles cannot be assigned Z coordinate values greater than 0.

5.3. Population Initialization

To provide the GA with a semi-ideal starting position, individuals in the GA are initialized according to the test cases generated by the usage model. If the individuals in the GA were initialized randomly, the GA would 'waste' generation cycles looking for individuals located within the user profile. Furthermore, with random initialization, it is possible that the GA may not find the individuals located in the user profile, and the results will be of little value. Because some of the individuals located in the user profile are already known, initializing the population with these known individuals can reduce the number of GA iterations.

5.4. Fitness Evaluation

The fitness of individuals is based primarily on maximizing two objectives, as graphically depicted in Figure 5. Optimal individuals are those that have a high likelihood of occurring and that result in failures with high failure intensity. Optimal individuals occur in zone 1. Inferior individuals are those with a low likelihood of occurring and would be located in zone 6.

While the GA system strives to find optimal individuals, there are two reasons that this is not always achievable. First, the software under test may be of such high quality that optimal individuals simply do not exist. Second, optimal individuals may exist outside of the defined user profile, but not within it. If the GA finds such individuals, they will be in zone 6 if they lie outside of the high usage areas of the software as defined by the usage model. Note that the boundary between the optimal, sub-optimal, and inferior zones is not necessarily a hard, distinct boundary. Since the user profile is simply an approximation for what the user may do, inferior individuals near the boundaries of the optimal and sub-optimal zones may also be of interest.

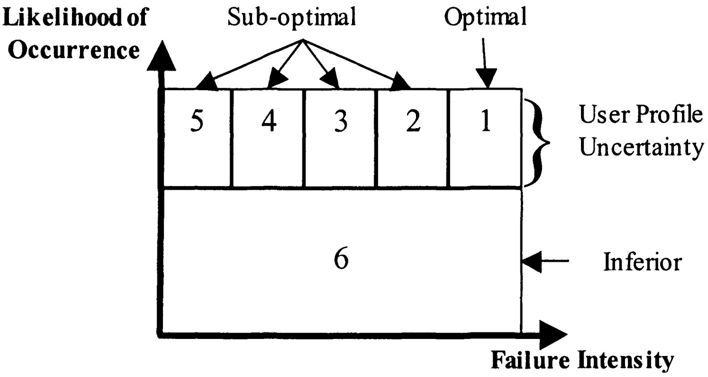


Figure 5. Fitness of Individuals.

The height of the optimal and sub-optimal zones is determined by the uncertainty in the accuracy of the user profile. If the user profile is based on historical evidence, or if the profile represents expert users, then the uncertainty in the accuracy of the user profile will be lower, resulting in a shorter height of the zones. However, if the user profile is based on guesswork, or if it represents novice users, then the uncertainty in the user profile accuracy will be higher, resulting in a taller height of the zones. The width and number of the optimal and sub-optimal zones is chosen according to the level of importance given to the GA concerning various levels of failure intensity. For example, if each failure were of equal importance, there would be only one optimal zone, no sub-optimal zones, and a width ranging from the lowest intensity level to the highest.

The overall fitness of an individual is based on likelihood of occurrence, the failures intensity, and the similarity to other individuals in the population. Each of these criteria is discussed in the following sections.

5.4.1. Likelihood of Occurrence

Individuals are first evaluated in terms of the likelihood they will be used by the user. Individuals containing input data that is very likely to be used by the user are very highly fit individuals for this particular objective. Individuals that contain input data that is not likely to be used by the user are very poorly fit individuals. This evaluation is based on the supplied user profile data set. The likelihood of the input data is calculated by multiplying the probability of occurrence of each input value that is used in the test case. For example, suppose the probability distribution for the input data is as shown in Table 5. The likelihood that the user would select Input Values 1 and 2 is 0.15. The likelihood that the user would select Input Values 1 and 3 is 0.0375. Consequently, a test case involving Input Values 1 and 2 would be rated as being more highly fit than one involving Input Values 1 and 3. The case study described in this paper only considers the first 4 genes in determining the likelihood of occurrence. This is because genes 1 – 4 provide the basics of the test scenarios while genes 5 – 10 provide the details. Likelihood of occurrence is based on the basics, not the details, of the scenario.

Input Value 1	0.75
Input Value 2	0.20
Input Value 3	0.05

Table 5. Input Data Probability Distribution

5.4.2. Failure Intensity

In addition to likelihood of use, the test team is also interested in test case individuals that find failures. Consequently, the second objective to be maximized is Failure Intensity, defined as a combination of failure density and failure severity. For example, suppose some individual causes a single failure that results in the crash of the software being tested. The Failure

Intensity consists of a low failure density (there is only 1 failure) and a high failure severity (the system crashes). In contrast, suppose another individual causes multiple failures that give erroneous output but do not crash the software being tested. In this situation, the Failure Intensity consists of a high failure density (there were multiple failures) and a low failure severity (the system does not crash, but gives erroneous output). Both of these individuals would be of interest, even though the composition of their Failure Intensity is different.

Consider the situation where a test manager differentiates failure severity according to five levels, with level 1 the lowest severity and level 5 the highest. For an individual test case that causes two level 3 failures, the failure intensity could be computed to equal 6, the sum of the failure severities. An individual that causes one level 5 failure would have failure intensity equal to 5. However, this approach to calculating failure intensity may not be satisfactory to the test manager. A single level 5 severity failure may be more important than a test case that produces multiple failures of lower severity. To handle this situation, a non-linear scoring method such as that shown in Table 6 is recommended.

Severity Level	Score
5	18
4	12
3	3
2	2
1	1

Table 6. Example Scoring Technique for Different Severity Levels.

If this scoring technique were applied, an individual that caused two level 3 failures would receive a failure intensity score of 6, and an individual that caused a single level 5 failure would receive an intensity score of 18. Similarly, an individual that caused three level 2 failures and two level 3 failures would receive an intensity score of 12. This yields a more useful result to the test manager than a linear scoring method. Obviously, the choice of scoring algorithm depends on the characteristics of the software being tested and the test management objectives.

5.4.3. Niching

As a genetic algorithm runs, the population of individuals will eventually converge to a single solution that dominates the population, and the diversity of the population is ultimately lowered. When a GA is applied to software usage testing, each individual represents a single test case. Consequently, the genetic algorithm would eventually converge to some test case that is both likely to occur and reveals failures of high intensity. To avoid having a single individual dominate the population, a niching technique [Holland 1975; Mahfoud 1995; Horn 1997] is used.

A niche represents some subpopulation of individuals who share some commonality. To apply this technique to software usage testing, a niche is formed for each unique combination of likelihood, failure intensity, and genetic values for the genes 1 through 4. That is, individuals that share the same likelihood, failure intensity, and genes 1 through 4 will occupy the same niche, or subpopulation. For example, a niche would be represented by a likelihood value of .07, a failure intensity value of 12, and genes values {2 2 3 6} for genes 1 through 4. In a population of 500, there may be 20 individuals who have these same values and would, consequently share this same niche. Another niche would be represented by a likelihood value of .05, a failure intensity of 10, and gene values {1 3 3 5} for genes 1 through 4. This type of niching is based on both the phenotype and partial genotype of the individuals. By implementing niches in the GA, the population will converge not to a single dominant individual, but to multiple dominant niches.

Specifically, niching is performed based on fitness sharing [Holland 1975]. Fitness sharing reduces the fitness values of individuals that are similar to other individuals in some way (i.e., the various niches in the population). This type of niching was used because of its success in prior work [Mahfoud 1995]. For this application, an individual's fitness value is reduced by dividing its fitness by the number of individuals that share its same niche.

5.4.4. Determining Overall Fitness

Highly fit individuals in the population are those maximize the objectives of likelihood of occurrence and failure intensity. A nonlinear aggregating fitness combination [Coello Coello et al. 2002] is used to identify individuals based on these two objectives. Determining failure intensity is already time consuming, therefore, this type of fitness combination was selected for its simplicity and speed. In addition, it directly addressed the needs of this particular case study.

Each individual *i* is given a combined fitness value that is based on the likelihood of occurrence of individual *i*, the failure intensity revealed by individual *i*, and the total number of individuals in the population *p* that also occupy the same niche as individual *i*. The fitness function to calculate the overall fitness value for an individual *i* is given as follows:

$$Fitness(i) = \frac{(Likelihood(i) \times Intensity(i))^{y}}{Niche\ Size(p, i)} \tag{1}$$

The variable y represents a nonlinear scaling factor that can be adjusted by the test team. This scaling factor is independent of the individuals in the population. Using the microscope analogy, the y value is analogous to the magnification level of the microscope. A higher y value represents a higher magnification, and vice versa. The higher the value of y used in the GA, the faster the population will converge to the most dominant niches, and the less diversity there will be in the population. The lower the value of y, the slower the population will converge and the more diversity there will be in the population (assuming that there is no one individual that is exceptionally fit).

If the scaling factor is not high enough, optimal individuals may not be found, or would be lost in the process. This may occur in large populations when weaker individuals may dramatically outnumber more optimal individuals. A higher scaling factor will help optimal individuals survive in a large mass of weaker individuals.

5.5. Genetic Operators

To create children from a given population, genetic operators such as selection, crossover, and mutation operators are applied to the individuals. Selection is first used to select parents from the population according to the overall fitness value, as discussed in section 5.4. Strongly fit individuals (higher fitness values) are more likely to be selected for reproduction than weaker individuals (lower fitness values). Consequently, the average population fitness should improve with each generation. Once parents are selected, crossover and mutation operators are applied to the parents to create children. The crossover and mutation operators provide the GA with the ability to explore the search space for new individuals and to create diversity in the population. The final result is a new population representing the next generation.

5.5.1. Selection

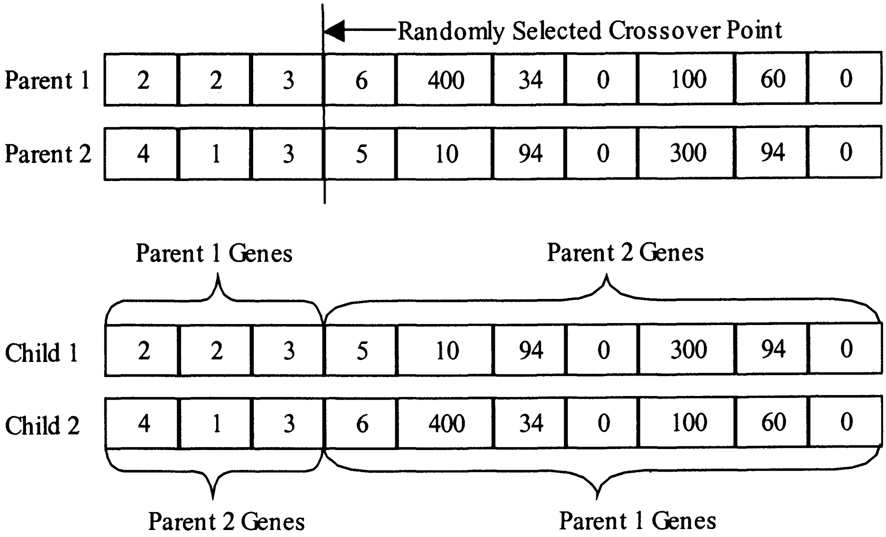
The GA selection process used for this application is the Fitness Proportional Selection [Holland 1975]. With this process, an individual's probability of being selected for reproduction is proportional to the individual's fitness with respect to the entire population. Each individual's fitness value is divided by the sum of the fitness values for all the individuals in the population. The resulting fitness value is then used to select parents, who then have the opportunity to pass on their genetic material (encoded information) to the next generation. Highly fit individuals are therefore more likely to reproduce. This helps to improve the quality of the population. An example of fitness proportional values is shown in Table 7. As can be seen, individual 4 is the most likely to be selected, and individual 2 is the least likely to be selected. Since this process depends on an individual's fitness proportional to the population, the tester can easily influence the selection process by altering the scaling factor of the fitness function, as discussed in section 5.4.4.

Individual	Original Fitness Value	New Fitness Value
1	2	2 / 21 = .0952
2	1	1 / 21 = .0476
3	4	4 / 21 = .1904
4	9	9 / 21 = .4285
5	5	5 / 21 = .2381
Sum	**21**	**.9998**

Table 7. Example of fitness proportional values.

5.5.2. Crossover

To create children, the GA for this application uses a single-point crossover operator that takes two parent individuals as input and outputs two children that are similar, but different, from the parents. This operator randomly selects a point in the genetic code of two parents and then swaps all genes between the parents that lie after the crossover point. When crossover is allowed between parents from different niches, diversity is encouraged. For this case study, every individual in each generation is processed by the crossover operator, and, if a child represents an invalid scenario, it is discarded from the population and replaced by its corresponding parent. For example, if Child 1 were invalid, it would be removed and replaced by Parent 1. The basic operation of crossover is shown in Figure 6.

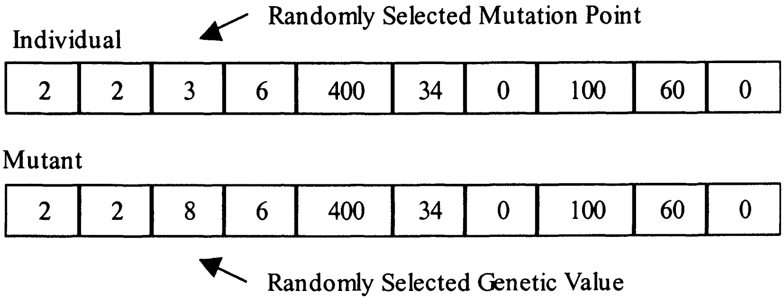


Parent 1	2	2	3	6	400	34	0	100	60	0
Parent 2	4	1	3	5	10	94	0	300	94	0
Child 1	2	2	3	5	10	94	0	300	94	0
Child 2	4	1	3	6	400	34	0	100	60	0

Figure 6. One-point crossover.

5.5.3. Mutation

In addition to the crossover operator, the GA for this application uses a single-point mutation operator that takes one individual as input, makes a small, random change to the genetic code of this individual, and outputs one mutant that is similar, but different, to the original individual. This operator randomly selects a gene in the genetic code of an individual and mutates that gene by randomly selecting some new value. For this case study, every individual in each generation is processed by the mutation operator, and, if the mutant represents an invalid scenario, it is discarded from the population and replaced by the original individual. The basic operation is shown in Figure 7.


Randomly Selected Mutation Point
Randomly Selected Genetic Value

Individual	2	2	3	6	400	34	0	100	60	0
Mutant	2	2	8	6	400	34	0	100	60	0

Figure 7. One-point mutation.

6. EXAMPLE

The application of the GA to software usage testing was based on a military simulation system. The population of interest for the examples included four terrain maps, thirty-seven primary and secondary actors, and seventeen tasks that are available for use with OTB.

To focus on observing and understanding the behavior of the GA for use in software testing, the Failure Intensity Evaluation portion of Figure 1 was simulated. Test cases were not actually performed on the military simulation system. A set of simulated failures was developed for use in all the examples. Simulated failures included problems with terrain maps, problems with a specific entity or task regardless of terrain, actor, etc. These simulated failures were representative of the types of problems seen in the real system. Failure intensities greater than 12 represented system crashes. Failure intensities less than 12 represented non-terminating failures. The scoring system used is shown in Table 8. This is the same scoring technique proposed in Table 6. Multiple failures per test case were also simulated. As a result, a test case may reveal a failure intensity of 5, meaning that there were two failures of with a score of 3 and 2, respectively.

Score	Meaning
18	Repeatable, terminating failure
12	Irregular, terminating failure
3	Repeatable, non-terminating failure
2	Irregular, non-terminating failure
1	No failures

Table 8. Failure intensity scoring system.

Two similar, but slightly different, user profiles were developed to examine the behavior of the GA when slight changes in a user profile occur. Sample test cases were generated for each user profile. The GA was initialized using each set of sample test cases, the corresponding user profile, and the input boundary (as described in section 5.1). For all the GA runs, the population size was 100 and the number of generations was 30.The results for three examples of the GA are shown in Figure 8, Figure 9, Figure 10, Figure 11, Figure 12 and Figure 13. Each point on the graphs represents a niche in the population, not a single individual. The data supporting these figures is shown in Table 9, Table 10, Table 11, Table 12, Table 13, and Table 14, respectively. These tables also show how many individuals occupy each niche.

In the first example, Figure 8 shows the niches that were formed after the fitness evaluation of the first generation formed from test cases generated according to User Profile 1. Figure 9 shows the niches that were formed after the fitness evaluation of the thirtieth generation. Notice that after 30 generations, the GA has converged to a few dominant niches. A comparison of Figure 8 and Figure 9 indicates that the GA has found four more niches that are very likely to occur and contain high failure intensities. Weaker niches did not survive.

In the second example, Figure 10 shows the niches that were formed after the fitness evaluation of the first generation formed from test cases that were generated according to User Profile 2. Figure 11 shows the niches that were formed after the fitness evaluation of the thirtieth generation. Notice that after 30 generations, the population of the GA has not converged sufficiently, but rather grew more divergent. This suggests that the fitness function and selection process are not sufficiently countering the effects of the crossover and mutation operators.

In the third example, the GA was reapplied using the same input data as in the second example. However, the scaling factor of the fitness function was increased from a value 1 to 2. This was done to increase the convergence of the population, so that the final population does not grow more divergent as in the second example. The initial niches for this example of the GA, shown in Figure 12, were the same as for the second example (i.e., Figure 12 is identical to Figure 10). However, as shown in Figure 13, the results were much different from that of Figure 11. These results are very similar to those shown in Figure 9. The GA has found four new niches that are very likely to occur and contain high failure intensities. The weaker niches did not survive.

The third example demonstrates a key aspect of the fitness function of the GA. The scaling factor of the fitness function plays a critical and delicate role in the finding and maintaining of optimal solutions. As illustrated in the second example, if scaling factor is too low, optimal solutions may not be found because the level of exploitation is diminished. However, if the scaling factor is too high, diversity and exploration will be diminished.

In the last two examples, the GA was able to overcome a less than optimal initial population. Notice in Table 11 and Table 13 that the initial populations were heavily biased towards the niche with the highest likelihood and low failure intensity. Table 12 and Table 14 show that the final populations are more balanced (in comparison to Table 11 and Table 13, respectively), and resulted in niches that are more interesting in terms of high failure intensity, while also being very likely to occur.

Finally, in each example, the final populations consist of niches that are:

1. Very likely to occur and resulted in a high failure intensity
2. Similar, but different. As described in [Myers 1979], similar, but different, test cases help to identify the failure's root cause.

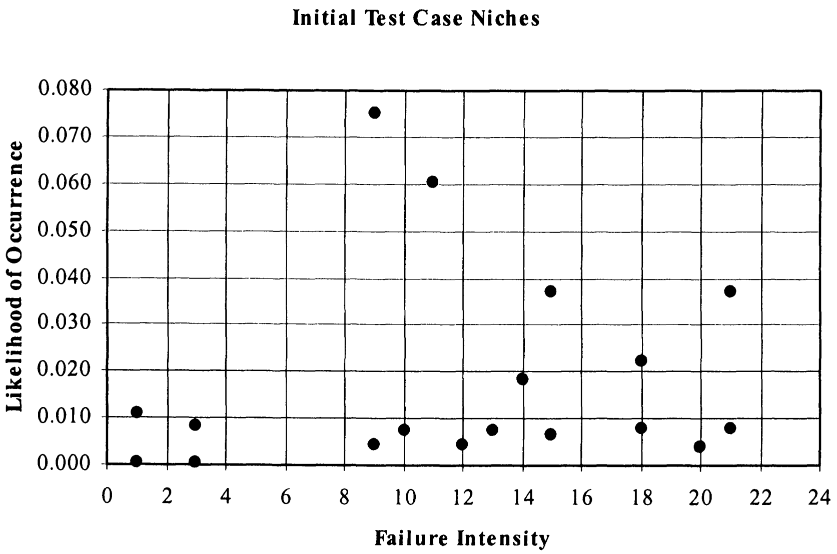


Figure 8. Test case niches for User Profile 1 after Generation 1.

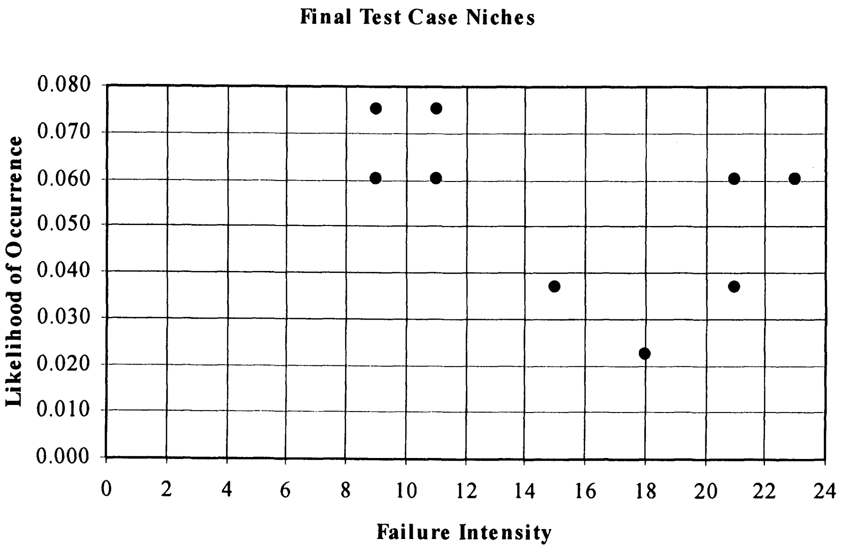


Figure 9. Test case niches for User Profile 1 after Generation 30.

Likelihood	Failure Intensity	Terrain	Primary Actor	Task	Secondary Actor	Niche Size
0.0005	1	Itsec	M16A2	Suppressive Fire	AK47	6
0.0005	3	Itsec	M16A2	Suppressive Fire	AK47	1
0.0040	20	Hunter	AH-64	Recon	SA-9	1
0.0040	20	Hunter	AH-64	Recon	SA-15	7
0.0043	9	Itsec	M16A2	Location Fire	AK47	5
0.0043	12	Itsec	M16A2	Location Fire	AK47	1
0.0067	15	NTC	M1A1	Assault	BMP-2	7
0.0072	10	Itsec	AH-64	Attack	T-72	3
0.0072	13	Itsec	AH-64	Attack	T-72	3
0.0080	18	Itsec	M3	Transport	SAW Gunner	5
0.0080	21	Itsec	M3	Transport	SAW Gunner	1
0.0083	3	Knox	M1A1	Assault	SA-9	7
0.0111	1	Knox	AC-130	Attack	SA-15	6
0.0185	14	Knox	AH-64	Recon	BMP-2	6
0.0223	18	NTC	M3	Transport	DI-M224	7
0.0370	15	Knox	AC-130	Ingress	SA-15	7
0.0370	15	Knox	AC-130	Ingress	SA-9	7
0.0370	21	Knox	AC-130	Ingress	T-80	7
0.0603	11	NTC	M1A1	Assault	T-72	7
0.0750	9	Knox	M1A1	Assault	T-80	6

Table 9. Number of individuals for test case niches shown in Figure 8.

Likelihood	Failure Intensity	Terrain	Primary Actor	Task	Secondary Actor	Niche Size
0.0223	18	NTC	M3	Transport	SAW Gunner	5
0.0223	18	NTC	M3	Transport	DI-M224	7
0.0370	15	Knox	AC-130	Ingress	SA-15	3
0.0370	15	Knox	AC-130	Ingress	SA-9	8
0.0370	21	Knox	AC-130	Ingress	T-80	8
0.0603	9	NTC	M1A1	Assault	T-80	6
0.0603	11	NTC	M1A1	Assault	T-72	5
0.0603	21	NTC	M1A1	Assault	T-80	19
0.0603	23	NTC	M1A1	Assault	T-72	13
0.0750	9	Knox	M1A1	Assault	T-80	11
0.0750	11	Knox	M1A1	Assault	T-72	15

Table 10. Number of individuals for test case niches shown in Figure 9.

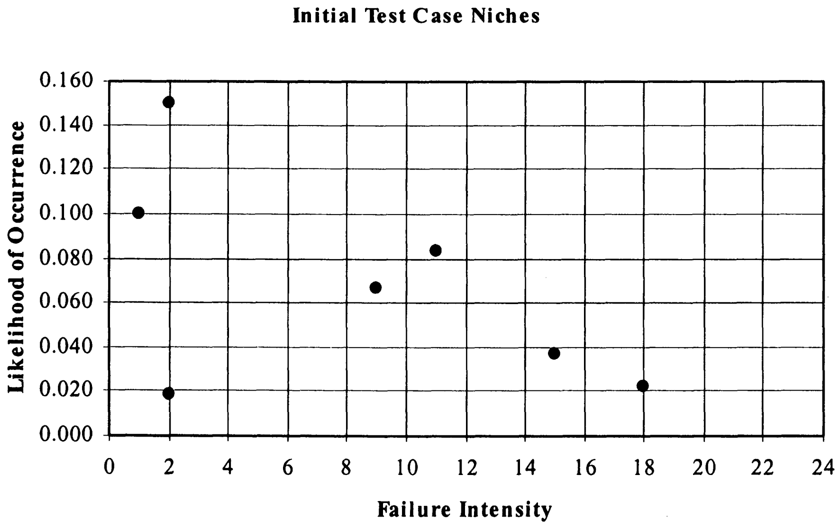


Figure 10. Test Case Niches for User Profile 2 after Generation 1.

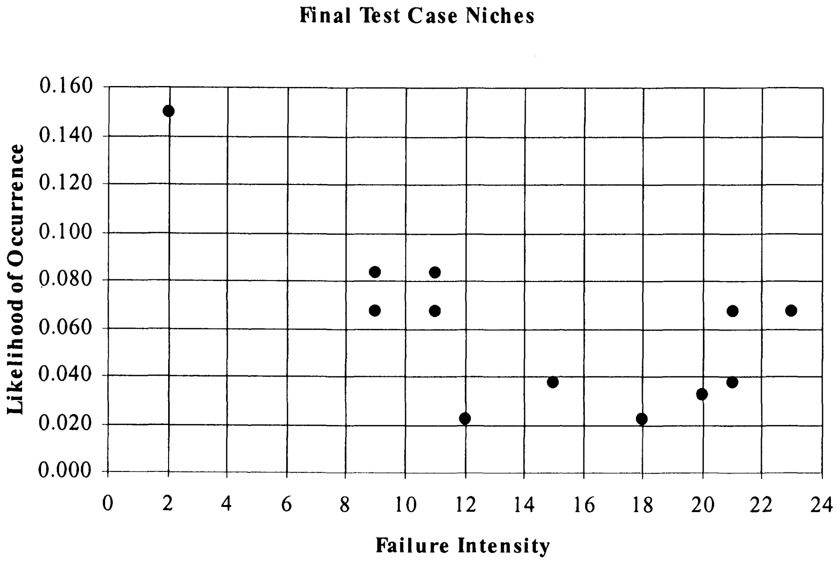


Figure 11. Test Case Niches for User Profile 2 after Generation 30.

Likelihood	Failure Intensity	Terrain	Primary Actor	Task	Secondary Actor	Niche Size
0.0185	2	Knox	AH-64	Recon	SA-9	8
0.0223	18	NTC	M3	Transport	SAW Gunner	8
0.0370	15	Knox	AC-130	Ingress	SA-15	7
0.0669	9	NTC	M1A1	Assault	T-80	8
0.0833	11	Knox	M1A1	Assault	T-72	8
0.0999	1	Knox	AC-130	Attack	SA-9	8
0.1499	2	Knox	AH-64	Attack	SA-9	53

Table 11. Number of individuals for test case niches shown in Figure 10.

Likelihood	Failure Intensity	Terrain	Primary Actor	Task	Secondary Actor	Niche Size
0.0223	12	NTC	M3	Transport	DI-M224	4
0.0223	18	NTC	M3	Transport	DI-M224	2
0.0321	20	Hunter	AH-64	Attack	SA-9	7
0.0370	15	Knox	AC-130	Ingress	SA-15	10
0.0370	15	Knox	AC-130	Ingress	SA-9	5
0.0370	21	Knox	AC-130	Ingress	T-80	4
0.0669	9	NTC	M1A1	Assault	T-80	9
0.0669	11	NTC	M1A1	Assault	T-72	8
0.0669	21	NTC	M1A1	Assault	T-80	18
0.0669	23	NTC	M1A1	Assault	T-72	13
0.0833	9	Knox	M1A1	Assault	T-80	5
0.0833	11	Knox	M1A1	Assault	T-72	9
0.1499	2	Knox	AH-64	Attack	SA-9	6

Table 12. Number of individuals for test case niches shown in Figure 11.

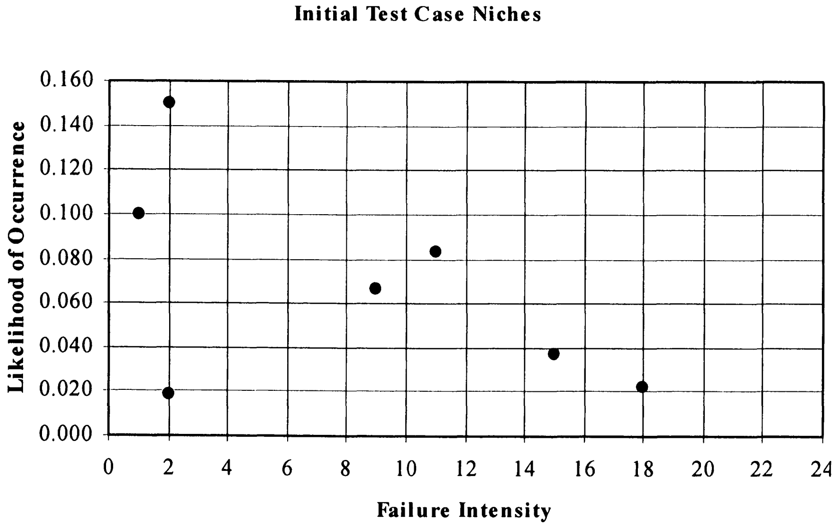


Figure 12. Test Case Niches for User Profile 2 after Generation 1 with scaling factor of 2.

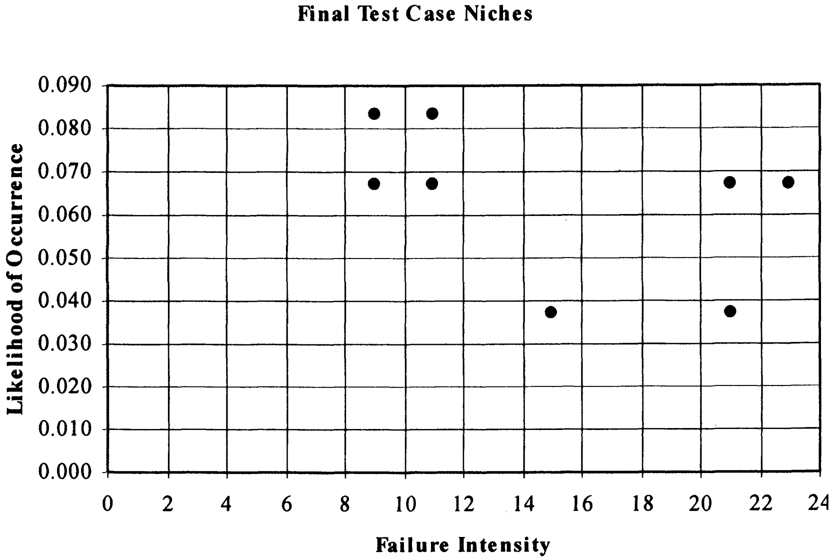


Figure 13. Test Case Niches for User Profile 2 after Generation 30 with scaling factor of 2.

Likelihood	Failure Intensity	Terrain	Primary Actor	Task	Secondary Actor	Niche Size
0.0185	2	Knox	AH-64	Recon	SA-9	8
0.0223	18	NTC	M3	Transport	SAW Gunner	8
0.0370	15	Knox	AC-130	Ingress	SA-15	7
0.0669	9	NTC	M1A1	Assault	T-80	8
0.0833	11	Knox	M1A1	Assault	T-72	8
0.0999	1	Knox	AC-130	Attack	SA-9	8
0.1499	2	Knox	AH-64	Attack	SA-9	53

Table 13. Number of individuals for test case niches shown in Figure 12.

Likelihood	Failure Intensity	Terrain	Primary Actor	Task	Secondary Actor	Niche Size
0.0370	15	Knox	AC-130	Ingress	SA-15	5
0.0370	21	Knox	AC-130	Ingress	T-80	7
0.0669	9	NTC	M1A1	Assault	T-80	2
0.0669	11	NTC	M1A1	Assault	T-72	9
0.0669	21	NTC	M1A1	Assault	T-80	30
0.0669	23	NTC	M1A1	Assault	T-72	31
0.0833	9	Knox	M1A1	Assault	T-80	6
0.0833	11	Knox	M1A1	Assault	T-72	10

Table 14. Number of individuals for test case niches shown in Figure 13.

7. CONCLUSIONS

This paper introduces a genetic algorithm approach to software usage testing that is used to explore the space of input data and identify and focus on regions that cause failures. Analysis of the examples in this paper demonstrates that genetic algorithms can be used as a tool to help a software tester search, locate, and isolate failures in a software system. The use of genetic algorithms supports automated testing and helps to identify those failures that are most severe and likely to occur for the user.

The strategy presented in this paper relies on a technique that not only helps the tester to isolate failure clusters, but also provides the developer with more information concerning the faults in the software and the input values that triggered them. The developer can then use this information to search, locate, and isolate the faults that caused the failures. The result can improve efficiency of both the testing and the development teams and can support subsequent improvements in the software development process.

The examples discussed in this paper raise a number of new ideas and issues for future consideration, such as the use of a global parallel genetic algorithm, different representation scheme, restrictive mating, and genetic algorithm parameter sensitivity to different user profiles. For example, current testing practice involves several testers working on different test cases at the same time. For the example application discussed in this paper, the fitness evaluation lends itself readily to parallelism. A global parallel genetic algorithm could take advantage of this parallelism. Such an approach could provide automated support to the current testing practice of distributed work effort. While each of these areas for future consideration could be further investigated with respect to applicability for software testing, as demonstrated by the examples of this paper, the simple genetic algorithm approach presented in this paper provides in itself a useful contribution to the selection of test cases and a focused examination of test results. Thus, application of this approach can support reasoning about test results to support quality system assessment and/or debugging activities.

ACKNOWLEDGEMENT

This work was funded in part by NAWC-TSD Contract N61339-01-D-002.

REFERENCES

Chambers, L., Ed. (2000), The Practical Handbook of Genetic Algorithms: Applications, Second Edition, Chapman & Hall / CRC.

Coello Coello, C.A., D.A.V. Veldhuizen, G.B. Lamont (2002), Evolutionary Algorithms for Solving Multi-Objective Problems, Kluwer Academic Publishers, New York, NY.

Coley, D. A. (2001), *An Introduction to Genetic Algorithms for Scientists and Engineers*, World Scientific, River Edge, NJ.

Deb, K. (1999), "Multi-objective genetic algorithms: Problem difficulties and construction of test problems", *Evolutionary Computation Journal*, 7,3, 205-230.

Dickinson, W., D. Leon, and A. Podgurski (2001), "Pursuing Failure: The Distribution of Program Failures in a Profile Space." In *Proceedings of the 9th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 246 – 255.

Drake, T (2000), "Testing Software Based Systems: The Final Frontier." *Software Tech News*, Vol. 3, No. 3.

Fonseca, C. M. and P. J. Fleming (1995), "An overview of evolutionary algorithms in multiobjective optimization", *Evolutionary Computation Journal*, 3,1, 1-16.

Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley.

Goldberg, D.E. (1990), "Real-coded Genetic Algorithms, Virtual Alphabets, and Blocking," Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois, Tech. Rep.90001.

Haupt, R. L., and S. E. Haupt (1998), *Practical Genetic Algorithms* John Wiley & Sons, Inc. New York, NY.

Holland, J. H. (1975), *Adaptation in Natural and Artificial Systems*. University of Michigan Press.

Horn, J. (1997), "The Nature of Niching: Genetic Algorithms and the Evolution of Optimal, Cooperative Populations", PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

IEEE Std. 829-1998, IEEE Standard for Software Test Documentation.

Karr, C. L., and L. M. Freeman, Ed. (1999), *Industrial Applications of Genetic Algorithms*, CRC Press, New York, NY.

Mahfoud, S. W. (1995), "Niching Methods for Genetic Algorithms." PhD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois.

Mitchell, M. (1996), *An Introduction to Genetic Algorithms*, MIT Press.

Myers, G.J. (1976), *Software Reliability*, John Wiley & Sons, Inc., New York.

Myers, G.J. (1979), *The Art of Software Testing*, John Wiley & Sons, Inc., New York.

Schultz, A. C., J. J. Grefenstette, and K. A. De Jong (1992), "Adaptive testing of controllers for autonomous vehicles." In *Proceedings of the 1992 Symposium on Autonomous Underwater Vehicle Technology*, pp. 158 – 164.

Zeller, A. (2001), "Automated Debugging: Are We Close?" *IEEE Computer*, 34, 11, 26-31.

An Expert System for Suggesting Design Patterns - A Methodology and a Prototype

David C. Kung, Hitesh Bhambhani, Riken Shah and Gaurav Pancholi

Department of Computer Science and Engineering
The University of Texas at Arlington
P. O. Box 19015, Arlington, TX 76019-0015
Tel: (817) 272-3627, Fax: (817) 272-3784
kung@cse.uta.edu

ABSTRACT

Software design patterns describe simple and elegant solutions to specific design problems. Design patterns capture design knowledge that have been discovered, evolved over time and proven to be effective in solving design problems. Application of design patterns improves software productivity and quality. Therefore the use of design patterns is rapidly increasing. However, it is not an easy task to choose an appropriate pattern to be applied from among the plethora of patterns. This is partly due to the learning curve involved to understand what each pattern can do for the designer. In this paper we present a methodology for constructing expert systems which can suggest design patterns to solve a designer's design problems. The methodology details the knowledge acquisition, knowledge representation and expert systems implementation activities. It is illustrated through the prototyping of the Expert System for Suggesting Design Patterns (ESSDP). Evaluation of the ESSDP by subjects other than the original developers indicates that the system indeed could suggest the needed design patterns effectively.

KEYWORDS

Design Patterns, Expert Systems, Object Oriented Design, Software Engineering.

1. INTRODUCTION

To cope with evolving requirements, software systems need to be flexible and easily updateable to incorporate the changes. This flexibility can be achieved by the use of appropriate design patterns. Design patterns provide a proven structure and characteristics for building highly maintainable and extendible software. According to [tich02a], the purpose of design patterns is to capture software design know-how and make it reusable. Design patterns can improve the structure of software, facilitate maintenance, and help avoid architectural drift. Design patterns also improve communication among software developers and empower less experienced personnel to produce high-quality designs. Design patterns make design more reusable besides capturing the know-how of design [tich98a]. Successful reuse of well-designed and well-tested software components improves software productivity, software quality and software reliability. Our experience in Internet software development also indicates that the use of design patterns significantly enhances the development team's ability to tackle complex design problems.

As per [alex77a], "each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". Patterns have roots in many disciplines, including plays, novels and most notably in Alexander's work on urban planning and building architecture.

The concept of design patterns is very much applicable to object oriented software. Software design patterns describe simple and elegant solutions to specific problems in object-oriented software design [gamm95a]. Design patterns capture solutions that have been discovered, developed and evolved over time. Each design pattern essentially has a pattern name, problem context, a general solution and related consequences. The pattern name is a noun phrase which summarizes the problem and solution and is easy to remember and communicate. For example, the singleton pattern provides an elegant solution to the problem of needing at most one globally accessible instance of an object like a database manager. Software developers then can refer to the name "singleton pattern" to communicate design ideas instead of having to repeat the problem and solution again and again.

Gamma et al classifies the twenty-three design patterns presented in their book [gamm95a] as:

- Creational Patterns, which comprise of Abstract Factory, Builder, Factory Method, Prototype and Singleton. These patterns provide various solutions for creating objects to serve different purposes.
- Structural Patterns, which comprise of Adapter, Bridge, Composite, Decorator, Facade, Flyweight and Proxy. These patterns provide solutions for composing or constructing larger structures that exhibit some desired properties. And
- Behavioral Patterns, which comprise of Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method and Visitor. These patterns deal with the algorithmic or behavioral issues of software design.

Use of software design patterns is rapidly increasing, propelled by the eagerness of the object-oriented community to enable efficient software reuse. It is of interest to note that the software industry agrees with the benefits of design patterns explained earlier [beck96a]. All six companies (FCS, AT&T, Motorola, BNR, Siemens and IBM) studied in [beck96a] agreed that design patterns are extracted from working models, capture design essentials and provide a good medium of communication. All but one company had the opinion that they enabled sharing of *best practices*. However, the roses have thorns and some companies had serious concerns regarding the challenges faced with design patterns. Some companies found that patterns are difficult and time consuming to write. Three of those industrial houses suggested that patterns should be introduced through mentoring. While a couple of companies felt that patterns require practice to write.

While the effective usage of design patterns is indispensable, finding out the most suitable one is a problem. One approach is to manually go through each one of them, or classes of patterns and narrowing down on a particular pattern. While this can be a chore for the novice in design patterns, experts can easily propose a design pattern suitable for the situation. The dearth of experts and the prohibitive costs to employ an expert designer pose a serious challenge to enterprises. The research by [schm96a] identifies a number of factors including organizational, economic, political and psychological factors that have been an impediment to wide spread use of design patterns. In addition to the above reasons, we feel that a salient impediment to their use is lack of flair for abstraction on the part of the developer. The situation worsens due to the exhaustive study that needs to be undertaken by the developer to conclude on the design patterns to be used in development.

To mitigate this problem one could hire an expert on design patterns or resort to an expert system which could suggest an appropriate software design pattern (SDP) to the developer. Expert systems are systems capable of offering solutions to specific problems in a way and level comparable to experts. According to [rile02a], “programs, which emulate human expertise in well-defined problem domains, are called expert systems”. Expert systems have been applied to solve problems in various domains.

In this paper we present a methodology to design and prototype an expert system for suggesting design patterns (ESSDP). The ESSDP selects a design pattern through dialog with the software designer to narrow down the choices. The dialog between the user and the system is a question and answer session, with the system posing questions and the user answering them. Classification and heuristics have been utilized to improve effectiveness.

Development of such an expert system involves initial knowledge acquisition, prototype development, prototype evaluation and continuous knowledge base refinement and enhancement [birm86a]. We have completed the initial knowledge acquisition, prototype development and prototype evaluation. Continuous refinement and enhancement is an ongoing activity. In our approach, the knowledge acquisition process represents acquired knowledge as trees where each node in the tree represents a circumstance under which a pattern could be applied. The proposed methodology details the process of formulating questions from these circumstances. To improve navigation efficiency, the questions are classified into various levels, each level contains questions with similar purposes. The methodology also enunciates a scheme to assign weights to the answers of the pattern-related questions. A pattern is selected once the weights of the answers for that pattern crosses the selection threshold value of that pattern. The proposed system was prototyped using CLIPS [rile02a]. The question and answer session was coded as rules. The rule-base had rules that were fired depending on the facts asserted by the user's answers. The system was validated successfully and found to be selecting appropriate patterns effectively.

This paper is organized as follows. The next section presents related work. Section 3 describes our five step methodology and section 4 the prototyping of ESSDP. A preliminary evaluation of the ESSDP system is given in section 5 which contains and explains the experimental results obtained from a small group of subjects who had used the system. The concluding remarks and future work are discussed in section 6.

2. RELATED WORK

By the time of writing, we have not found publications that report on expert systems for suggesting design patterns. Therefore, this section mainly reviews some work related to research and applications of design patterns.

Gamma et al's book [gamm95a] is a classic book on design patterns. It contains a comprehensive treatment of twenty-three commonly used patterns, classified into creational patterns, structural patterns and behavioral patterns. Nobel [nobl98a] discusses relationships between design patterns, that is, a pattern uses another pattern, a pattern refines another pattern or a pattern conflicts with another pattern. Also mentioned are relationships stating that a pattern is similar to another pattern or one pattern is combed with another pattern. The mapping of the latter to the former is demonstrated in the paper.

Tichy [tich98a] provides a catalogue for over 100 general-purpose design patterns. Tichy categorizes all design patterns into nine broad categories, which are termed as "top level categories". These nine categories are decoupling, variant management, state handling, control, virtual machine, convenience patterns, compound patterns, concurrency and distribution. It is suggested in the paper that categories should be mutually exclusive (exceptions may be allowed in some cases), and subcategories are strict subsets of the parent categories and should be mutually exclusive as far as possible.

Many developers have considered design patterns for development of software, and focused on understanding design patterns. Monroe [monr97a] explored the capabilities and roles of design patterns, architectural styles and objects and their strengths and limitations. Riehle [Rieh96] discussed the crucial aspects of the pattern concept, relate patterns to the different models and pattern forms to help developers understand and apply patterns. Tan [tanh99a] proposed to apply patterns to solve problems that occur in database applications such as reuse of transaction specifications and external error handling. The patterns proposed can be applied during database design and design verification. Herrmann [herr99] described the application of OO software design within the CHAMP project. "Abstract Factory" and "Façade" patterns were applied. Enhancement of flexibility and the limitations were discussed with implementation examples. In [schm96a], the author described how design patterns were applied on a number of large-scale commercial distributed systems as well as ways to avoid common traps and pitfalls of applying design patterns.

Re-engineering of existing software using design patterns to improve reusability, maintainability and understandability is discussed by Keller [kell99a]. An environment for re-engineering of design components based on the structural descriptions of design patterns were described using three case studies. Like Keller, William Chu [chuw99a] has used the parallel program generation environment (PPGE) as a case study to the re-engineering of a traditional software system into a pattern based software system. The result achieved is that it is better to re-engineer legacy systems rather than re-designing them as re-engineering is more cost effective and less risky. Masuda [guom00a] has described the application of software design patterns for redesigning existing software. The evaluation of the resulting software and the existing software is accomplished using C&K metrics [chid94a]. This paper shows that the application of design patterns to software greatly enhances its flexibility and makes it more easy to extend.

3. THE METHODOLOGY

In this section, we describe our approach for developing expert systems that can suggest design patterns. In particular, we will describe the methodology that we used to develop the Expert System for Suggesting Design Patterns (ESSDP). ESSDP is a tool that selects a design pattern based on the user's requirements. ESSDP engages the user in a question-answer session that helps narrow down the selection process. At the end of the process a suitable design pattern is suggested with a certainty. The ESSDP is a selection type of expert system and hence we choose to use a rule-base as the knowledge base of choice to develop ESSDP. Besides, a rule-based expert system has been proven to be useful due to its ease of development, extension and enhancement. The rules that form the rule-base are if-then conditionals that capture the expert knowledge [pede89a].

ESSDP is aimed to help a beginner to find/decide on which pattern to use for a particular design situation. It can also be used to validate a designer's choice of a particular design pattern. Use of ESSDP can be integrated well with standard software processes.

Software development process generally iterates through the stages of specification, analysis and design, implementation, testing and evolution. A widely accepted incremental and iterative process is the Unified Process (UP) [jacob99a]. The UP is architecture centric, use case driven and acknowledges the risks involved. It spreads the software development activity among requirements, analysis, design, implementation and testing workflow. For each increment (a portion of the system with a sub-set of the functionality) one iterates the process involving the above five workflows. Usually each workflow includes a few iterations before satisfactory outputs are produced for the next workflow. In particular, during the design workflow the system analyst or designer produces a first cut of the classes and refines it until the requirements for that increment are fulfilled by the classes. We recommend that the system analyst/designer consider use of the ESSDP after the first cut (iteration). After the first iteration a designer is more aware of the design problems/issues and is in a better position to answer the questions posed by ESSDP, thus enhancing the benefits from this tool. An interesting observation made by Hunt [hunt00a] is that during the initial iterations of system design majority of the problems are related to architectural/structural issues while the later iterations deal with problems that are more behavioral. ESSDP can provide suggestions for architectural, structural and behavioral design problems. Hence it is useful throughout the design workflow over various increments.

The methodology for developing an expert system like ESSDP consists of five iterative steps, summarized as follows:

- Step 1. Identify circumstances in which a pattern can be applied.
- Step 2. Refine the circumstances with sub conditions.
- Step 3. Formulate questions to ask the user.
- Step 4. Classify the questions according to their levels of significance.
- Step 5. Assign thresholds to patterns and weights to questions.

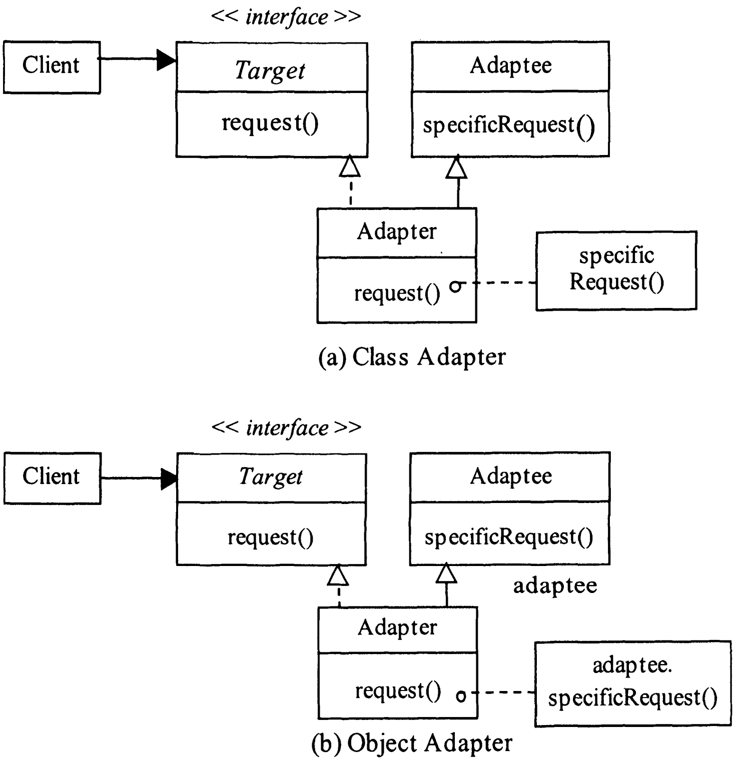
After step 5, the results are implemented using an expert system shell. This will be described in the next section (Prototyping). Here we describe the five steps in greater detail in the following subsections:

3.1. Step1. Identify Circumstances In Which A Pattern Can Be Applied

Prior to constructing the rule-base we perform knowledge acquisition. Knowledge acquisition refers to the task of gathering the required knowledge and verifying it. During this phase literature on object-oriented software design patterns was reviewed and analyzed. For prototyping ESSDP, we limit our knowledge acquisition to literature survey because we feel that the literature on design patterns has contained rich knowledge that is adequate for our initial effort. The expert system designer may interview human experts to obtain knowledge on design patterns. The authors serve as knowledge engineers as well as experts in the field of design patterns. The expertise among the authors is at varied levels with one of the authors possessing academic and industrial experience of patterns. The other three authors, which include two masters and one doctoral student, have practiced patterns in an industry sponsored project and have reviewed literature on patterns.

For the prototype we select the design patterns in Gamma's book [gamm95a]. Each design pattern is analyzed and its characteristics are identified. For every design pattern, we formulate a set of circumstances in which the pattern can be used. From the set of circumstances, at least one circumstance has to occur for that design pattern to be applied. Some patterns may require more than one circumstance to be present. These circumstances are conditions for the applicability of that design pattern. Henceforth we will refer to these circumstances as conditions and vice versa.

An example of knowledge acquisition by the above method can be explained using the adapter pattern. The adapter pattern, by its name, makes the interface of one component or object conform to the other by delegating the request from the client to the real subject. Figure 1 illustrates the adapter pattern in the Unified Modeling Language (UML). It lets classes with incompatible interfaces work together [gamm95a].



(a) Class Adapter

(b) Object Adapter

Figure 1. Class adapter and object adapter in UML.

Figure 1 shows two adapter patterns, one class adapter and the other object adapter. The class adapter implements an interface which the client wants to see. The adapter also subclass to the Adaptee class and hence inherits the superclass' behaviors. A request from the client is fulfilled by calling the appropriate method (specificRequest() in the figure) that is inherited from the superclass. Thus, the class adapter adapts the interface of the Adaptee class to the interface that the client wants. Similarly, the object adapter does the same except that it adapts the interface of a particular instance of an object class rather than the interface of a class. In this case the object adapter delegates the request from the client to an instance of the Adaptee class.

From the description of the adapter pattern [gamm95a], we can derive the following list of conditions:

```
A1: Want to use an existing implementation.
A2: Desired interface is different from the existing
    interface.
```

This set of conditions can be depicted graphically as a tree shown in Figure 2. Note that A1 alone is not sufficient for suggesting the adapter pattern but A1 and A2 would be.

3.2. Step 2. Refine The Circumstances With Subconditions

As a result of Step 1 we obtain various trees similar to Figure 2 for each pattern. In step 2 we refine each condition into possible subconditions. This is done for each tree from Step 1. A sub-condition is depicted as a child of a condition in the tree. This is illustrated in the Figure 3. In the Figure we see that A1 has three children A11, A12 and A13, while A2 has A13 and A21 as its children. Further, A11 has A111 as its child.

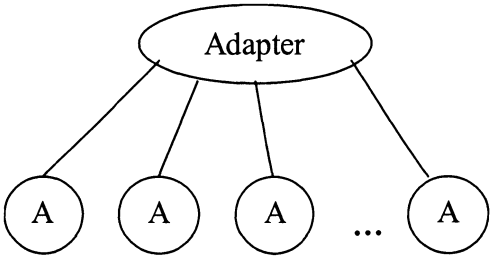


Figure 2. An initial circumstance tree from the Adapter pattern.

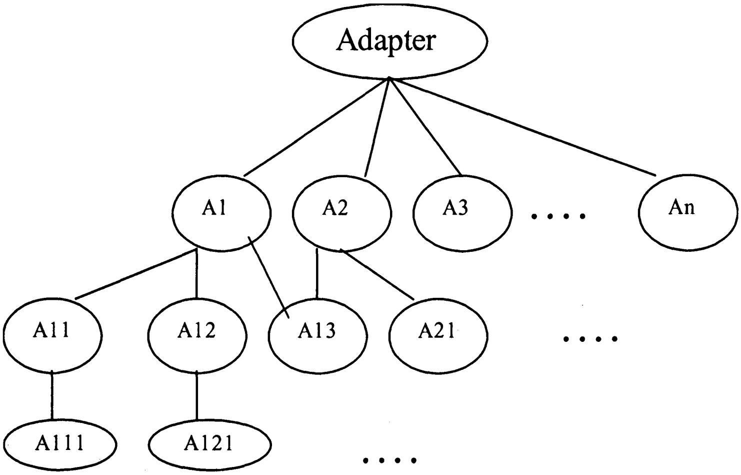


Figure 3. The circumstance tree for the Adapter pattern.

From Step 1 we had A1 to A2 as the conditions needed for the adapter pattern to be applied. The child conditions for these three conditions are the subconditions. Each child condition describes the possible circumstances for the occurrence of its parent condition. This can be explained by looking at the refinements of condition A1 (Want to use an existing implementation) which are:

```
A11: Want to save time and effort.
A12: Re-implementation of an existing class is costly.
A13: Need to use third party software.
```

Each child condition of A1 is a condition for A1 to occur. It can be interpreted as: "wanting to use an existing implementation" could be due to "needing to use third party software", another reason could be that the existing class is costly to re-implement, etc. Similar refinement is carried out for each node in the tree, unto a considerable level.

Notice that A13 is a subcondition for A1 (Want to use an existing implementation) as well as A2 (Desired interface is different from the existing interface). This implies that A13 is also a subcondition for A2. In other words one of the reasons for A2 is the presence of third party software, which is denoted by A13.

Similarly each pattern tree is refined to add subconditional nodes.

3.3. Step 3. Formulate Questions To Ask The User

As a result of Step 1 and Step 2 we obtain the tree representation[1] of knowledge about design patterns. That is, a set of trees with various levels of conditions represented as nodes. These trees are the means that help us derive the questions that are asked of the user. The answers to these questions get asserted as facts in the knowledge base. These facts in turn may fire some rules. The rules either suggest a pattern or ask more questions, when enough information is not yet-available.

In this step, we convert each node from the tree to a question, preferably, answerable by either yes or no. For example, the node A1 (Want to use an existing implementation) from Figure 3 can be framed as the following question "Do you want to use an existing implementation?". Similarly, node A13 (Third party software) becomes "Is a part/class of your system a third party software?"

The result of this step is a set of questions formulated for each tree generated from Step 2.

3.4. Step 4. Classify The Questions According To Their Levels Of Significance

During this phase we partition the various questions into different levels. Each level then consists of a few questions. The questions in each level are directed towards reducing the search space to select a design pattern. The questions are divided among the following five levels:

- Level 0: pattern category selection questions
- Level 0.5: pattern subcategory selection questions
- Level 1: intent questions
- Level 2: pattern specific questions
- Level 3: auxiliary questions

Each of these levels is explained in greater detail in the following subsections.

3.4.1. Level 0: Pattern Category Selection Questions

This level contains questions that narrow down the search space to a particular classification of patterns. In accordance to [gamm95a], the twenty-three design patterns are classified into creational patterns, structural patterns and behavioral patterns. Creational patterns deal with creation or instantiation of objects. Structural patterns describe structural compositions of classes and objects. Behavioral patterns provide ways to assign responsibilities to objects and designing the communications between various objects and their interconnections [gamm95a].

The user's response to a Level 0 question guides the system to focus on questions relating to the patterns in the group selected by the user. As an example, for the ESSDP to distinguish between creational, structural and behavioral patterns we present the user with the following questions:

```
L0Q1: Is your design problem concerned with:
      creating complex objects -or-
      architectural structures of classes -or-
      behavioral aspect of objects -or-
     don't know?
```

Selection of an option from L0Q1, asserts a corresponding fact in the system. This fact determines which other questions to be asked of the user. The first option asserts a fact that steers the system towards questioning related to creational patterns. While the second option shifts focus to questions related to structural patterns and the third option towards behavioral patterns. Finally, the last option indicates that the user cannot choose. In this case, the system will ask Level 0.5 questions (see section 3.4.2) from all categories, resulting in more questions need to be asked. The strategy we have used to develop the ESSDP is to ask a couple of questions from each of the Level 0.5 categories and narrow down the search space according to the users answers to these questions. A positive answer enhances the likelihood to pursue that category and a negative answer reduces the likelihood.

3.4.2. Level 0.5: Pattern Subcategory Selection Questions

The questions at this level are an auxiliary to the purpose of level 0 questions. That is, the questions at level 0.5 also help narrow down the search space to a classification of patterns. This is done through classification of patterns within each category of level 0 patterns, that is, the categories of creational patterns, structural patterns and behavioral patterns. Consider for example the seven structural patterns from [gamm95a]: adapter, bridge, composite, decorator, facade, flyweight and proxy. As discussed earlier the adapter provides a uniform abstraction of various interfaces by making one interface conform to the other. The bridge pattern lets one change the implementation without having to change the abstract interface. The composite pattern describes formation or compositions of objects in part-whole hierarchies; thus, it allows clients to treat each object uniformly. The decorator pattern allows a program to dynamically add or remove functionality to objects. This is achieved by describing an alternative to subclassing to extend the object's functionality. The facade pattern helps create a simple, unified interface to a subsystem or a group of subsystems. The flyweight pattern describes how to share large number of objects efficiently. The proxy pattern is used to provide a surrogate for controlled access to an object [gamm95a].

To partition the seven structural patterns the expert system designer must analyze the patterns to find partition criteria. Various partition criteria can be defined, depending on the set of patterns at hand. For the seven structural patterns, we found that some of them dealt with interface issues while the others dealt with complex structures to enhance functionality, security and/or efficiency. Thus, we partition the seven structural patterns into two groups, the first group having four patterns and the second group three. The first group named interface consists of adapter, bridge, facade and proxy patterns. These patterns are grouped together since each one of them deals with issues relating to the interface of classes or objects. The second group named complex consists of composite, decorator and flyweight patterns. The patterns in this group deal with complex architectures of not just a couple of objects but various objects.

The question that is formulated then is:

```
L0.5Q1: Is your design problem concerned with
        component interfacing -or-
        constructing a complex component through
        composition?
```

Figure 4 depicts the hierarchy of patterns resulting from above steps.

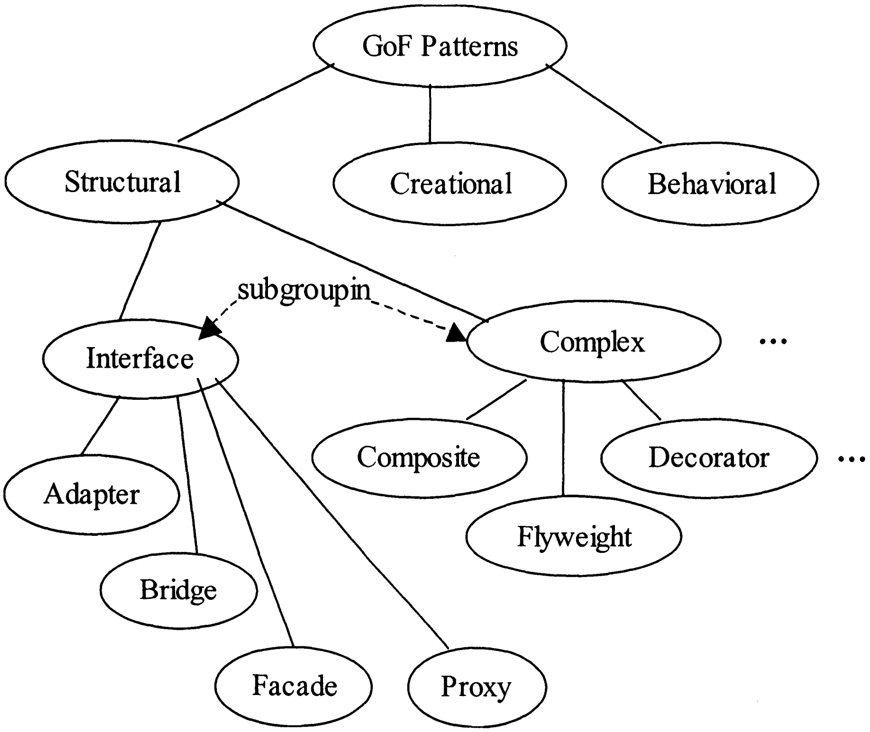


Figure 4. A classification of design patterns.

As is evident from figure 4, the patterns are grouped as creational, structural and behavioral patterns. The questions at level 0 guide the search of patterns towards one group out of the three. In structural pattern hierarchy there is another subgrouping of interface and complex model patterns. The questions at level 0.5 reduce the focus of search to even fewer patterns present in the subgroup. The answers from the questions that constitute level 1, 2 and 3 finally guide the expert system to suggest a design pattern. These levels are explained in detail below.

3.4.3. Level 1: Intent Questions

The questions of level 1 are derived from the intent or main idea of each pattern. The intent of each pattern is used to formulate one question for each pattern. Care is taken that the question represents the gist of the pattern, nothing more and nothing less.

This step derives most of its questions from the set of questions that resulted from step 3. Below are some example questions that qualify as level 1:

L1Q1: Is your design problem concerned with adapting one interface to another?

L1Q2: Is your design problem concerned with notifying other objects when an object changes?

L1Q3: Is your design problem concerned with state dependent behavior?

L1Q4: Is your design problem concerned with selecting algorithms according to needs?

The question L1Q1 summarizes the adapter pattern, L1Q2 the observer pattern, L1Q3 the state pattern and L1Q4 the strategy pattern. The adapter pattern has already been explained earlier (recall in section 3.1 that "A1: Want to use an existing implementation" alone is not sufficient to suggesting the adapter pattern but A1 and "A2: Desired interface is different from the existing interface." would be sufficient.). The observer pattern defines a many-to-one relation between objects, allowing observer objects to be notified if the state of a "publisher" is changed. A commonly used example is a collection of data to be displayed as a pie chart, bar chart and a table. The collection of data is the publisher while the charts and table are the observers. The state pattern allows the object to modify its behavior depending on its internal state. A typical application of the state pattern is implementation of a state machine. The strategy pattern allows a client to select one algorithm from a family of algorithms. For example, the strategy pattern may be used to select different sorting algorithms to satisfy different sorting needs.

3.4.4. Level 2: Pattern Specific Questions

The questions in level 2 are pattern specific questions and are derived from the conditions for applying each pattern. Every question created in step 3 for each pattern is reviewed to determine whether it is level 2 or not. There are some necessary conditions that need to be present for a pattern to be selected for use. Generally, level 2 questions are the ones that result from these conditions. Such conditions can be traced back to the tree structure for a pattern (Figure 3), where these conditions are the children of the root. Since these conditions are specific for the pattern to be of use, we call these questions pattern specific questions.

From our analysis of the adapter pattern, the following questions qualify as level 2:

Adapter-L2Q1: Do you have classes that have incompatible interfaces?

Adapter-L2Q2: Do you want to adapt an implementation to a desired interface?

3.4.5. Level 3: Auxiliary Questions

The questions in level 3 are pattern specific but less important questions and are derived from the subconditions for each pattern. We call this set of questions the auxiliary questions to help confirm the suggestion of a particular pattern. Every question created in step 3 for each pattern is reviewed to determine which level it should be put in. Of those, the questions that qualify as level 3 are mostly the less important ones. Usually the questions that are in level 3 are the ones that remain after the selection of level 2 questions. These questions are formulated from the subconditions that were added to pattern trees in step 2. These questions are less important than the ones in level 2 and act more as auxiliary information.

For the adapter pattern, the following questions are level 3:

```
Adapter-L3Q1: Do you want to use an existing
              implementation?
Adapter-L3Q2: Is re-implementation of existing classes
              costly?
Adapter-L3Q3: Is a part/class of your system third-party
              software?
Adapter-L3Q4: Do you want to avoid re-implementation of
              classes?
```

It is quite possible for ESSDP to conclude on a pattern depending on the answers of the pattern specific questions from level 2. To accommodate novice and advanced users, we provide an option whereby ESSDP can present an early conclusion or it can continue till questions are exhausted. The auxiliary or level 3 questions tend to be simpler or more intuitive, making them easier for the novice user to understand.

The pool of questions generated after step 3 may contain questions that are common for two or more patterns. During the creation of the rule-base care is taken to ensure that the common questions are asked only once.

3.5. Step 5. Assign Thresholds To Patterns And Weights To Questions

Once all the questions are determined and categorized in levels 0, 0.5, 1, 2 and 3, we assign a selection threshold and a rejection threshold to each pattern. A selection threshold for a pattern is a positive integer and is defined as the minimum number of points required for a pattern to be selected as the answer. The selection threshold is reached by the accretion of weights assigned to answers for level 2 and level 3 questions. The rejection threshold is a negative integer and is defined as the maximum number of points required to still pursue the pattern. When the tally of points drops below the rejection pattern, the ESSDP abandons that pattern and stops asking questions related to that pattern.

Prior to deciding the thresholds for each pattern, weights are assigned to the questions. The weight of a question is a positive integer and signifies the importance of that question towards selection of that pattern. Hence the level 2 questions have more weight as compared to the level 3 questions. The sum total of the weights for all the questions of a pattern is 100. A positive answer to a question adds the weight of that question to a positive-response-counter. A negative-response-counter is reduced by the weight of the question when the user enters a negative reply. Questions related to the pattern are exhausted when the difference between the positive-response-counter and the negative-response-counter is 100. At this point the selection threshold criterion is evaluated to decide whether the pattern satisfies the user requirements. During the evaluation of the answer of each question, ESSDP compares the negative-response-counter and the rejection-threshold. In the case that the counter has dropped below the threshold, the pattern is removed and further questioning related to that pattern stops.

The selection-threshold and rejection-threshold values have to be determined after consultation with a group of experts or from extensive review of literature. For prototyping ESSDP the authors served as the group of experts and reached consensus on the assignments of the weights and thresholds based on their experiences with design patterns.

A structured method, such as the Delphi process [lins75a], involving a group of estimators, could be applied to determine the weights and the thresholds. The Delphi process works as follows. First, each estimator independently suggests his weightings and thresholds. The estimates are then compiled and each estimator explains her/his reasoning behind her/his estimate. This is done only for estimates that are significantly different. The estimators then re-estimate and the process is repeated until they reach a consensus.

As shown in the next section and in Table 1, the ESSDP system can suggest a design pattern whenever the positive-response-counter is greater than the selection-threshold. The user then is given the option to continue the dialog to increase the certainty value of the suggestion or accept the suggestion and terminate the process. This early conclusion feature can be toggled on or off according to the user's preference.

To illustrate this procedure, we present a high level overview of the interaction between the system and the user along with the rule-base action. This is presented in a tabular manner below (see Table 1).

#	System Response	Pattern-seeker action	ESSDP action
1.	Is your design problem concerned with creating complex objects (create) – or – architectural structure of classes (struct) - or – behavioral aspect of objects (behav) – or – don't know(dontknow) :	struct	Structural pattern selected.
2.	Is your design problem concerned with component interfacing (iface) - or - constructing a complex object through composition (complx)?	iface	Interface patterns selected
3.	Is your design problem concerned with adapting one interface to another? (yes/no)	yes	Adapter pattern intended
4.	Is your design problem concerned with keeping implementation and interface independent of each other? (yes/no)	no	No action
5.	Is your design problem concerned with providing a simple and easy to use interface for a subsystem? (yes/no)	no	No action
6.	Is your design problem concerned with controlling access to an object? (yes/no)	no	No action
7.	Do you have classes that have incompatible interfaces? (yes/no)	yes	Positive-response-counter = 0 + 26 = 26
8.	Do you want to adapt an implementation to a desired interface? (yes/no)	yes	Positive-response-counter = 26 + 26 = 52
9.	Do you want to use an existing implementation? (yes/no)	yes	Positive-response-counter = 52 + 12 = 64
10.	Is re-implementation of existing classes costly? (yes/no)	yes	Positive-response-counter = 64 + 12 = 76
11.	Suggesting Adapter Pattern with 0.76 certainty. Would you like to continue with more questions? (yes/no)	yes	Continue
12.	Is a part/class of your system third-party software? (yes/no)	no	Negative-response-counter = 0 - 12 = -12
13.	Do you want to avoid re-implementation of class? (yes/no)	yes	Positive-response-counter = 76 + 12 = 88
14.	Suggesting Adapter Pattern with 0.88 certainty.		

Table 1. User-ESSDP interaction to suggest the Adapter pattern.

As is shown in the table, the system starts with a level 0 question. A fact is inserted in the rule-base depending on the answer. In the example shown above, the fact asserted is Structural-patterns-selected. Similarly user action in row 2 results in Interface-patterns-selected being asserted as a fact. From row 3 the user affirms the intent question related to adapter pattern, hence a corresponding fact is asserted in the rule-base. This steers the system towards asking the user level 2 and level 3 questions related to the adapter pattern. The questions in row 7 and row 8 are the level 2 questions while the questions in rows 9 to 13 are from level 3.

Let us say that level 2 questions have been assigned a weight of 26 points while level 3 questions have 12 point weight each. With a total of 2 questions in level 2 and 4 questions in level 3, we have the required total of 100 points. For the example let us put the selection-threshold to be 70 points and the rejection-threshold to be 30.

As shown in rows 7 and 8, a positive answer to the level 2 question increases the positive-response-counter by 26 points. Hence by the end of level 2 round of questions, the positive-response-counter is 52 points. A yes answer to any of the level 3 questions results in the positive-response-counter being incremented by 12 points. In row 10, the positive-response-counter has reached 76 which is greater than the selection-threshold 70. Therefore, the system suggests the adapter pattern with 0.76 certainty in row 11 and let the user choose if to continue with more questions. In row 12 the negative-response-counter is decreased by 12 points because the user gives a negative answer to the question.

The last question related to adapter is the one in row 13. The rule-base finds that no more patterns are intended to be pursued and the positive-response-counter is more than the selection threshold of 70 points. This eventuates in a fact being asserted indicating that a pattern has been concluded. The fact further clarifies that it is adapter pattern.

4. PROTOTYPING

In this section we describe the implementation of ESSDP resulting from the methodology described earlier. We choose to use CLIPS, version 6.10, as the expert system language. CLIPS, an acronym for C Language Integrated Production System [rile02a], is a multi paradigm programming language. That is, CLIPS supports rule-based, object oriented and procedural programming [giar94a]. We describe below the architecture of the software system for ESSDP and the CLIPS rule-base.

4.1. Software Architecture

The ESSDP has four major components, as shown in Figure 5. The components are the rule-base, the inference engine, the fact list and a user interface. The rule-base is a collection of rules that are needed by the expert system to deduce facts and finally suggest a design pattern. The ESSDP rule-base for the prototype was generated using the five-step methodology described above. The rule-base was programmed into CLIPS using the *defrule* construct provided by the language. This will be discussed in detail in the next section.

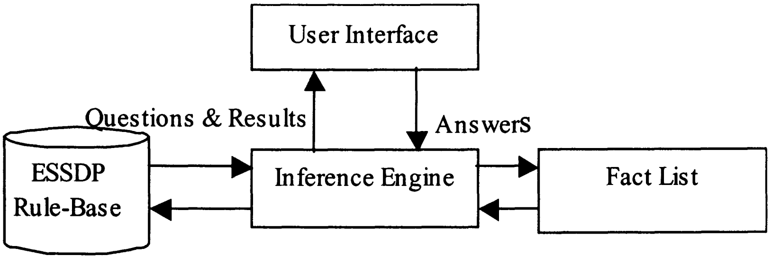


Figure 5. ESSDP Software architecture.

The fact list contains the data on which inferences are derived. It is a part of the CLIPS environment and is maintained by the inference engine. It consists of the current facts in the system.

An inference engine is that part of the expert system that is already programmed and ready for use. It interprets the knowledge bases, which in our case is a rule-base, and controls the overall execution. CLIPS consists of a forward chaining inference engine. A forward chaining inference engine is an algorithm that derives new facts from the ones already present in the system, ultimately reaching a conclusion. The CLIPS inference engine maintains a fact-list and uses this list to match patterns against the current state of the fact-list. The matching of patterns determines which rules are ready for execution. It also resolves conflicts when several rules are applicable at the same time. The inference engine interacts with the user through the user interface to ask questions and receive answers.

The user interface to ESSDP allows the user to answer multiple-choice questions and questions that need a 'yes' or a 'no' reply. For ESSDP the user interface is a simple, text-oriented interface. Interaction with a user is accomplished by the functions available in CLIPS for input/output. We have implemented a Web based version of ESSDP but in this paper we choose to present only the text oriented interface.

4.2. The ESSDP Rule-Base

A rule is comprised of two parts, the antecedent and the consequent. The antecedent is the *if* portion of the rule and is a set of conditions or facts that must be satisfied to execute the rule. In CLIPS, the antecedent conditions are satisfied by the existence or non-existence of facts in the fact-list. The consequent is the *then* portion of a rule and comprises of actions that are performed when the rule is executed. The CLIPS inference engine executes all the action statements whenever the antecedent of a rule is satisfied.

The five-step methodology described earlier helps us produce the questions, categorize them in various levels of functionality and also assign weights to them. We then write rules that present those questions to the user and process the answers along with the rules to determine the outcome.

CLIPS also provides us mechanisms to assign priorities to rules. It is called salience in CLIPS. When multiple rules are present in the agenda a rule with higher salience fires before a rule with lower salience. Since we have rules that ask questions and rules that suggest patterns, we assign them different salience. All question rules are assigned the normal salience, which by default is 0. Rules that suggest patterns are assigned a negative salience number so that they are fired after the firing of question rules. To represent a pattern in CLIPS, we define a record for a pattern. A record is similar to a struct in C and defines a data structure that can refer to as a whole. The *deftemplate* construct is used in CLIPS to define such a record. The record is called pattern and has following fields:

- **Name:** The name of the pattern.
- **Category:** The category of the Pattern. This is more like a flag, which holds integer values to indicate the category of that pattern. The categories can be any one of structural, behavioral, creational, interface or complex.
- **Final:** This is an integer flag which indicates whether a conclusion has been reached for that pattern.
- **Certainty:** The certainty for the recommendation of a particular pattern.
- **Asked-List:** This list is used to track which questions have already been presented to the user.

When the user starts the ESSDP, a fact is asserted which is (start-questions). This fact does not directly contribute towards determining the design pattern, however it does help in controlling the flow of decision-making and in the question-answer session. We term such facts as control-facts.

Rule 1: This rule is named as *zero-stage-division* and asks the user the level 0 question. Note variables are denoted by names beginning with a question mark (?) in CLIPS.

```
IF there is (start-questions)
AND there is NOT (concluded pattern)
begin
  /*
    ask-question-function prints the question on the
    screen and returns the answer from the user in the
    variable
  */
  ?ans <= ask-question-function (
    "Is your design problem concerned with:
    creating complex objects (create) -or-
    architectural structures of classes (struct) -or-
    behavioral aspect of objects (behav) -or-
    don't know (dontknow)?"
  )
  if ?ans = struct then assert (pattern (name
  "structural") (level 0))
  if ?ans = behav then assert (pattern (name
  "behavioral") (level 0))
  if ?ans = create then assert (pattern (name
  "creational") (level 0))
  if ?ans = dontknow then
  begin
    assert (pattern (name "structural") (level 0))
    assert (pattern (name "behavioral") (level 0))
    assert (pattern (name "creational") (level 0))
  end
  /* remove the fact start-questions, i.e. change mode
  of ESSDP */
  retract (start-questions)
  /* control fact to indicate ESSDP mode is now asking-
  questions */
  assert (asking-questions)
end
```

In the above rule, we use the record structure called *pattern*. It gets asserted as a fact in the system with some or all of the fields filled. After each question is asked, the asked-list is updated.

Rule 2: The rule shown below is named *first-stage-struct-division*. It helps the user decide which subgroup to pursue. In case the category of patterns does not have subgroups then such a rule is omitted.

```
IF there is (asking-questions)
AND there is NOT(concluded pattern)
AND there is (pattern (name "structure") (level 0))
  begin
  ?ans <= ask-question-function(
    "Is your design problem concerned with component
    interfacing (iface) -or-
    constructing a complex component through composition
   (complx) -or-
    don't know (dontknow)?"
   )
  if ?ans = iface then assert (pattern (name
  "interface") (level 1))
  if ?ans = complx then assert (pattern (name "complex")
  (level 1))
  if ?ans = dontknow then
```

```
    begin
      assert (pattern (name “interface”) (level 1))
      assert (pattern (name “complex”) (level 1))
    end
  end
```

Rule 3: Given below is a rule that asks the intent questions, that is level 1 questions from the user. The example rule shown below asks questions related to interface type patterns, which are adapter, bridge, facade and proxy.

```
IF there is (asking questions)
AND there is NOT (concluded pattern )
AND there is (pattern (name “interface”) (level 1))
begin
  ?ans <= ask-question-function (
    “Is your design problem concerned with adapting one
    interface to another? (yes/no)”)
  if ?ans = yes then  assert (pattern (name
Adapter”)
  (level 99))
  ?ans <= ask-question-function (
    “Is your design problem concerned with keeping
    implementation and interface independent of each
    other? (yes/no)”)
  if ?ans = yes then assert (pattern (name “Bridge”)
  (level 99)))
  ?ans <= ask-question-function(
    “Is your design problem concerned with providing a
    simple and easy to use interface for a subsystem?
    (yes/no)”)
  if ?ans = yes then assert (pattern (name “Facade”)
  (level 99))
  ?ans <= ask-question-function (
    “Is your design problem concerned with controlling
    access to an object? (yes/no)”)
  if ?ans = yes then assert (assert (pattern (name
  “Proxy”) (level 99)))
end
```

Similar rules are produced for complex type patterns. In case of patterns with no sub grouping, we can directly ask intent questions (level 1) without having to go through the phase of asking level 0.5 questions.

Rule 4: Once all the rules related to level 0, level 0.5 and level 1 are written, we create rules to ask level 2 and level 3 questions. As an example, shown below is a rule for question two of adapter level 2.

```
IF there is (asking questions)
AND there is NOT (concluded pattern)
AND there is (pattern (name “Adapter”) (level 99) (final
0))
```

```
AND there is NOT (pattern (name “Adapter”) (level 99)
(final 0)
asked-list($?prefix L2-2 $?suffix))
begin
  ?ans <= ask-question-function(
    “Do you want to adapt an implementation to a
    desired interface?(yes/no)”)
  if ?ans = yes then
    ?adapter-positive-response-counter <= ?adapter-
    positive-response-counter + ?adapter-level2-weight
    if ?adapter-positive-response-counter > adapter-
    selection-threshold then
      modify (pattern (name "Adapter") (final 1))
  else
    ?adapter-negative-response-counter <= ?adapter-
    negative-response-counter - ?adapter-level2-weight
  if ?adapter-negative-response-counter < ?adapter-
  rejection-threshold then
    retract (pattern (name “Adapter”) (level 99) (final
    0))
    modify (pattern (name “Adapter”) (asked-list insert
    L2-2))
end
```

The multivalued variables are denoted by names beginning with the $? symbol in CLIPS.

In the above rule, the pattern record for adapter is modified to indicate the final field is 1 when the positive counter exceeds the selection threshold.

Rule 5: This rule suggests a design pattern with a certainty value to the user whenever the positive counter accumulates to the selection threshold. The user is then asked whether to continue with more questions or stop and accept the suggested design pattern. This rule can be turned on and off as an option to suite different users' preferences. The rule below is shown for the adapter pattern.

```
IF there is (asking questions)
AND there is NOT (concluded pattern)
AND there is NOT (adapter mid selection over)
AND there is (pattern (name "Adapter") (level 99) (final
1)
(certainty ?cert))
begin
  ?ans <= ask-question-function(
    “Suggesting adapter with certainty value of ?cert
    Would you like to continue with more questions?
    (yes/no)”
  )
  if ?ans = yes then
    modify (pattern (name "Adapter") (level 99) (final
    0))
  else
    modify (pattern (name "Adapter") (level 99) (final
    2))
  assert (adapter mid selection over)
end
```

If the user does not want to continue, then the final flag is set to 2. The system will skip the questions and activate the rule shown below. A final value of 0 indicates that the user wants to continue.

The rule that decides the final answer is shown below. It has a salience of -999 so that it fires only after all the rules described above have been fired.

```
IF there is NOT (pattern (name ?pname) (level 99) (final
0))
AND there is (pattern (name ?name) (level 99) (final 2)
(certainty ?cer))
AND there is NOT (concluded pattern)
begin
  assert (concluded pattern ?name ?cer)
end
```

Another rule then prints out the final answer on the user interface depending on the details asserted in (concluded pattern) fact. The last rule in the rule-base is one that has the lowest salience of -1000. This last rule fires on the absence of (concluded pattern) fact and displays a "failed to choose a pattern" message to the user.

All of the rules described above are coded in a CLIPS file. This CLIPS file, apart from the rule-base, also contains the initial-facts to get the system started, the threshold values and the user interface functions. The system, when fully implemented, will consists of about 160 rules, 150 questions and 3,000 lines of CLIP code excluding comments.

5. EVALUATION

We have conducted a small experiment to assess the effectiveness of ESSDP. The experiment involved 11 students from a design patterns class at the beginning of the course. The students, referred to as subjects, were given an overall introduction to what design patterns could do and design problems the patterns could solve. The subjects then were asked to randomly select ten out of the eighteen patterns (seven structural patterns and eleven behavioral patterns) that had been implemented at the time of the experiment. The subjects were required to use ESSDP to search for the patterns that could solve their design problems. The subjects had to submit the dialogs with the expert system and provide information on the subject themselves and evaluations of three aspects of the ESSDP as shown in Table 2. We consider the experiment to be partial and preliminary due to the small number of subjects participated and the lack of detailed assessment.

More extensive assessments will be conducted when the system is fully implemented and optimized. However, this preliminary, initial assessment is considered useful for the following reasons:

1. It helps us in determining if the effort is worthwhile. The initial feedback as shown by the experiment result (see below) is very encouraging.
2. It helps us to assess whether the ESSDP system can be used by software engineers other than the ESSDP developers. The feedback is very positive. Most of the subjects or all of the subjects we asked said that the system was friendly and very easy to use.
3. It helps us identify weaknesses and improvements. For example, although the number of questions needed to be answered is acceptable in most cases, reducing the number will effectively improve the usefulness of the system.

Subj	OO Knowledge	DP Knowledge	Success Rate	Effectiveness	#Questions
1	beginner	want to know	all the time	very effective	12
2	beginner	know some	most of the time	effective	12
3	knowledgeable	don't know	all the time	very effective	17
4	knowledgeable	want to know	most of the time	effective	18
5	knowledgeable	know some	all the time	very effective	9
6	knowledgeable	know some	most of the time	effective	10
7	knowledgeable	don't know	most of the time	effective	12
8	knowledgeable	know some	all the time	very effective	12
9	knowledgeable	know some	all the time	very effective	15
10	beginner	don't know	all the time	very effective	13
11	knowledgeable	know some	all the time	very effective	8
Summary					
	knowledgeable 73%	know some 55%	always 64%	very effective 64%	less than 13 64%
	beginner 27%	don't know 45%	most of the time 36%	effective 36%	more than 12 36%

Table 2. Summary of testing of ESSDP by 11 subjects.

A brief explanation of Table 2 is as follows. The OO Knowledge column denotes how good the subject knew OO concepts and OO design. The table shows that 73% of the subjects believed that they were knowledgeable, only 27% were beginners. Knowledgeable is interpreted as an average rating, meaning they know OO concepts but are not experts or "know a lot". Only 55% knew some design patterns before joining the class. 45% did not know any or wanted to know what were design patterns. Data in these two columns are consistent. The lower percentage in the "DP Knowledge" column than the "OO Knowledge" column reflects the fact that DP knowledge requires OO knowledge as a prerequisite.

```
Is you desing problem concerned with
creating complex objects, (create)
-or-
architectural structures of classes (struct)
-or-
```

```
behavioral aspect of objects (behav)
-or-
don't know(dontknow). Struct

Is your design problem concerned with component
interfacing (iface)  -or-
constructing a complex component through composition
(complx)? Complx
Is your design problem concerned with recursive
composition of complex objects from simpler ones?
(yes/no) yes

Is your design problem concerned with dynamically adding
responsibilities to objects? (yes/no) no

Is your design problem concerned with representing
numerous copies of
the same object? (yes/no) no

Do you want your system to be layered, by grouping
components? (yes/no) yes

Would you like to make your system easier to add
components? (yes/no) yes

Do you want to hide difference between composite objects
and
individual objects from client application? (yes/no) yes

Will you treat composite objects uniformly? (yes/no) yes

Do you want to represent part-whole hierarchy? (yes/no)
yes

******************************************************
******************************************************

Suggesting Composite pattern, with 1.0 certainty.
```

Figure 6. Search scenario 1: the user knows what he wants.

```
Is you desing problem concerned with
creating complex objects, (create)
-or-
architectural structures of classes (struct)
-or-
behavioral aspect of objects (behav)
-or-
don't know(dontknow). dontknow

Is your design problem concerned with
component interfacing (iface)  -or-
constructing a complex component through composition
```

```
(complx)? Complx

Is your design problem concerned with notifying other
objects when an object changes (yes/no) no

Is your design problem concerned with state dependent
behaviors? (yes/no) no

Is your design problem concerned with selecting
algorithms according to needs? (yes/no) no

Is your design problem concerned with undoing of
operations? (yes/no) no

Is your design problem concerned with accessing contents
of an aggregate object without exposing its internal
representation ? (yes/no) yes

Is your design problem concerned with recursive
composition of complex objects from simpler ones?
(yes/no) yes

Is your design problem concerned with dynamically adding
responsibilities to objects? (yes/no) no

Is your design problem concerned with representing
numerous copies of the same object? (yes/no) no

Will you traverse through a list of aggregate-object in
different ways depending upon your motive ? (yes/no) yes

Do you want to provide a uniform interface for traversing
different aggregate structures ? (yes/no) no

Do you want to provide a way to browse through aggregate
objects ? (yes/no) yes

Do you want your system to be layered, by grouping
components? (yes/no) yes

Would you like to make your system easier to add
components? (yes/no) yes

Do you want to hide difference between composite objects
and individual objects from client application? (yes/no)
yes

Will you treat composite objects uniformly? (yes/no) yes

Do you want to represent part-whole hierarchy? (yes/no)
yes

******************************************************
```

```
*******************************************************

Suggesting Iterator pattern, with 0.6 certainty.
```

Figure 7. Search scenario 2: the user does not know what he wants.

The last three columns, “Success Rate”, “Effectiveness” and “#Questions (answered before getting a suggestion)” consistently show that 64% of the subjects considered the system to be able to suggest the needed patterns all the time, very effective and require the user to answer no more than twelve questions. The individual subject assessments also individually confirmed the result. That is, subjects who had “all the time” rating for “Success Rate” also had “very effective” rating for “Effectiveness” and most of them required to answer about twelve questions (except subject 3 and subject 9 who had to answer seventeen and fifteen questions, respectively). It is worth noticing that the (evaluation of the) performance of the system is independent to the degrees of OO knowledge and design pattern (DP) knowledge of the subjects. This independence confirms to our initial motivations of developing the ESSDP. That is, providing pattern search assistance to novices as well as validating pattern beginner's choices.

In summary, the preliminary evaluation indicates that the system is able to suggest the needed patterns most of the time, is effective and requires answering of no more than twelve questions. The number of questions needed to be answered will be reduced if the system evaluates the thresholds each time the thresholds are updated and recommend the pattern whose positive counter exceeds the positive threshold. We have prototyped such changes and found that the number of questions needed to be answered has reduced by 20% or 10 questions needed to be answered.

We wish to point out that the more the user knows his design problem the more effective the search will be, as illustrated in Figure 6 and Figure 7. In Figure 6, the user selects “struct” as the answer to the first question while in Figure 7 the user selects “dontknow” as the answer.

In the first case, the user knows that his design problem is concerned with composing classes/objects to form larger structures while in the second case the user does not know this. The first case requires the user to answer ten questions and suggested the Composite pattern with a 1.0 certainty (100%). However the second case requires the user to answer eighteen questions and finally the system suggested the Iterator pattern with only a 0.6 certainty (60%). That is the number of questions needed to be answered by the user increased by 80% while the certainty of the pattern suggested decreased by 40%. In this case, we are not clear if the pattern suggested would solve the user's design problem. But the low certainty value of 0.6

suggests that it is not likely that the iterator pattern would solve the user's design problem.

6. CONCLUSIONS

We have presented a five step methodology for constructing an expert system for suggesting design patterns and illustrated the methodology through the construction of the ESSDP expert system. ESSDP implements the twenty-three design patterns in Gamma et al's book. Our preliminary evaluation of ESSDP by eleven subjects shows that ESSDP and hence the methodology are relatively effective, although much improvements are required.

As near term future work, we plan to refine the ESSDP system to improve its success rate and effectiveness and reduce the number of questions needed to be asked of the users. We also plan to extend the system to cover more design patterns including analysis patterns [fowl96a], responsibility assignment patterns [larm01a] and patterns from [tich98a]. Patterns from specific domains like telecommunications design patterns will also be considered.

We have implemented a Web portal for accessing the ESSDP system. We plan to integrate the two to provide WWW access to the expert system so that OO developers from other organizations can benefit from the work. Providing a WWW access will enable us to collect feedback from a large users base with various applications and use feedback to further improve the system.

ACKNOWLEDGEMENT

We would like to thank Larry Holder for suggesting the CLIPS expert system shell and resources. We also like to thank the anonymous reviewers for the improvement suggestions and Chien-hung Liu for help on CLIPS.

NOTES

[1] It is more appropriate to call it a lattice rather than a tree but we prefer to use tree since it is easier to explain the methodology using trees.

REFERENCES

[alex77a] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King and S. Angel, "A Pattern Language," Oxford University Press, New York, 1977.

[beck96a] K. Beck, J. O. Coplien, R. Crocker, L. Dominick, G. Meszaros, F. Paulisch and J. Vlissides, "Industrial Experience with Design Patterns", In Proceedings of the 18th International Conference on Software Engineering (ICSE '96), IEEE Computer Society, Los Alamitos, CA, 1996. pp. 103-114.

[birm86a] W. Birmingham, R. Joobbani and J. Kim, "Knowledge Based Expert Systems and Their Applications", In Proceedings of the 223rd ACM/IEEE Conference on Design Automation, ACM Press, New York, NY, 1986. pp. 531-539.

[chid94a] S. Chidamber and C.F. Kemerer, "A metrics suite for object-oriented design," IEEE Trans. on Software Engineering, vol. 2, no. 6, pp. 476-493, June 1994.

[chuw99a] W.C. Chu, Chih-Wei Lu, J.P. Shiu, X. He, "Pattern Based Software Re-engineering: A Case Study," IEEE Proceedings of the Asia Pacific Software Engineering Conference, 1999. pp. 300-308.

[fowl96a] Martin Fowler, "Analysis Patterns," Addison Wesley Longman, Inc. 1996.

[gamm95a] Erich Gamma, Richard Helm, Ralph Johnson and John M. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995.

[giar94a] J. Giarratano, G. Riley, "Expert Systems," Second Edition, PWS Publishing Company, Boston, MA, 1994.

[guom00a] Masuda Gou, Sakamoto Norihiro, Ushijima Kazuo, "Redesigning of an Existing Software using Design Patterns," Proc. of Int. Sympo. On Principles of Software Evolution 2000, pp.169-173.

[hunt00a] John Hunt, "The Unified Process for Practitioners", Springer-Verlag New York, Inc., New York, NY, 2000.

[jacob99a] I Jacobson, G. Booch and J. Rumbaugh, "The Unified Software Development Process.", Addison-Wesley, Reading, MA, 1999.

[kell99a] R.K. Keller, R. Schauer, S. Robitaille, P. Page, "Pattern-Based Reverse-Engineering of Design Components," ICSE'99 Los Angeles CA, 1999.

[lins75a] H.A. Linstone and M. Turoff. "The Delphi Method: Techniques and Applications," Addison-Wesley, Reading, MA, 1975.

[larm01a] Craig Larman, "Applying UML and Patterns," Prentice Hall, 1998.

[monr97a] R.T. Monroe, A. Kompanek and D. Garlan, "Architectural styles, design patterns, and objects," IEEE Software, Vol. 14 No. 1, Jan.-Feb. 1997. pp. 43--52.

[nobl98a] J. Noble, "Classifying relationships between object-oriented design patterns", In Proceedings of Australian Software Engineering Conference (ASWEC), IEEE Computer Society, Los Alamitos, CA, 1998. http://citeseer.nj.nec.com/noble98classifying.html.

[pede89a] K. Pedersen, "Expert Systems Programming: Practical Techniques for Rule-Based Programming," John Wiley & Sons, New York, NY, 1989.

[rile02a] Gary Riley, "A Tool for Building Expert Systems", http://www.ghg.net/clips/CLIPS.html.

[schm96a] D. C. Schmidt, "Using Design Patterns to Guide the Development of Reusable Object-Oriented Software", ACM Computing Surveys, 28(4es): 162, ACM Press, New York, NY, 1996.

[tanh99a] Hee Beng Kuan Tan, Tok Wang Ling, "Integrated design patterns for database applications," Journal of Systems and Software, July 1999, pp. 159-172.

[tich98a] W. F. Tichy, "A Catalogue of General-Purpose Design Patterns", In Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS 23), IEEE Computer Society, 1998. pp. 330-339.

[tich02a] W. F. Tichy, "Essential Software Design Patterns", http:// wwwipd.ira.uka.de/~ tichy/patterns/overview.html, 2002.

Condensing Uncertainty via Incremental Treatment Learning

Tim Menzies[1], Eliza Chiang[2], Martin Feather[3], Ying Hu[2], James D. Kiper[4]

1 Lane Department of Computer Science
University of West Virginia, Morgantown, USA
tim@menzies.com

2 Electrical & Computer Engineering University of British Columbia
yingh@ece.ubc.ca
echiang@interchange.ubc.ca

3 Jet Propulsion Laborator
California Institute of Technology
martin.s.feather@jpl.nasa.gov

4 Computer Science & Systems Analysis
Miami University
kiperjd@muohio.edu

ABSTRACT

Models constrain the range of possible behaviors defined for a domain. When parts of a model are uncertain, the possible behaviors may be a data cloud: i.e. an overwhelming range of possibilities that bewilder an analyst. Faced with large data clouds, it is hard to demonstrate that any particular decision leads to a particular outcome. Even if we can't make definite decisions from such models, it is possible to find decisions that reduce the variance of values within a data cloud. Also, it is possible to change the range of these future behaviors such that the cloud condenses to some improved mode. Our approach uses two tools. Firstly, a model simulator is constructed that knows the range of possible values for uncertain parameters. Secondly, the TAR2 treatment learner uses the output from the simulator to incrementally learn better constraints. In our incremental treatment learning *cycle, users review newly discovered treatments before they are added to a growing pool of constraints used by the model simulator.*

1. INTRODUCTION

Often, during early lifecycle decision making in software engineering, analysts know the *space* of possibilities, but not the *constraints* on that space. For example:

- They might know qualitatively that the more shared data, the less modifiable is a software system. However, they may not know the exact quantitative values for this relationship.
- Their experience might tell them that their source lines of code estimates are inaccurate by 50%.

What are our analysts to do? One possibility is to demand more budget and time to perform further analysis which removes these uncertainties. For example, metrics collection programs might be commenced to collect values for uncertain parameters. Elsewhere, we have documented the impressive results that can come from such a methodology/process [30].

When elaborate metrics collection is too expensive however, computational intelligence methods may be useful. If domain experts can offer a rough description of how (e.g.) variable A effects variable B, then fuzzy logic methods [17, 55] can be used to perform inference over the model, perhaps using the methods of Jahnke et al. [25]. If the model represents a situation for which we have historical data, then genetic algorithms can be used to mutate the current model towards a model that best covers the historical data [3]. Alternatively, we could throw away the current model and use the historical data to auto-generate a new neural net model [50].

The premise of this paper is *metrics starvation*; i.e. situations in which we can access neither the relevant domain expertise required for fuzzy logic, nor the historical data required for genetic algorithms or neural nets. Our experience is that metrics starvation is common. For example, the majority of software development organizations do not conduct systematic data collection. As evidence for this, consider the Software Engineering Institute's capability maturity model (CMM [43]), which categorizes software organizations into one of five levels based on the maturity of their software development process. Below CMM level 4, there may be no systematic and reliable data collection. Below CMM level 3, there may not even be a written definition of the software process. Many organizations exist below CMM level 3 [personnel communication, SEI researchers]. Hence, reliable data on SE projects is scarce or hard to interpret.

However, a lack of systematic data does not mean that no inferences can be made about some software development process. If we can't constrain the range of model behavior with domain metrics, we can still make decisions by surveying the range of possible behaviors. Suppose we have a model expressing what is known within a domain. If we are uncertain over parts of that model, then we might supply *ranges* for those uncertain parameters. When we run this model, if we ever require some uncertain parameter, we might *select* and *cache* a value for that parameter, based on the supplied ranges. To survey the range of possible behaviors, we just parameters. Elsewhere, we have documented the impressive re-run the model many times, taking care to clear the cache between each run. Many variants on this scheme have been discussed in the literature. For example:

- This scheme is the same as Monte Carlo simulations when uncertain parameters are just system inputs.

- This scheme is the same as abductive inference [27] where the uncertain parameters are truth assignments to assumptions within the model, and some global invariant checking executes before a new value is assigned.

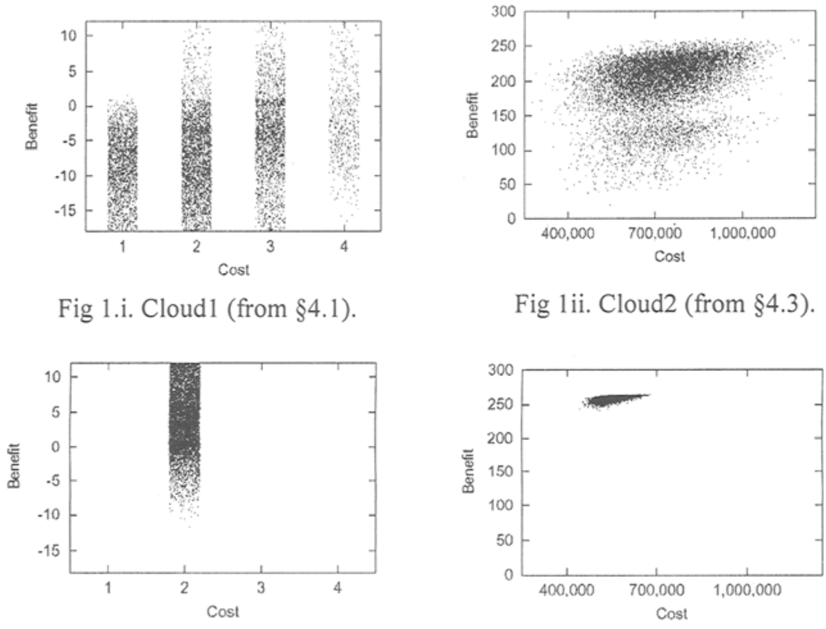


Fig 1.i. Cloud1 (from §4.1).

Fig 1ii. Cloud2 (from §4.3).

Fig 1.iii. Cloud1, condensed.

Fig .iv. Cloud2: condensed.

Figure 1. Examples of condensing clouds. The right-hand model's cost values are continuous while the left-hand model has discrete costs.

The advantage of this "select and cache'" method is that the range of possible behaviors can be explored *without* expensive further analysis. The disadvantage of this approach is *data clouds*: an overwhelming amount of data that clouds and confuses the issues. For example, Figure I.i and Figure I.ii show data clouds generated from case studies described later in this paper. In these figures, each mark represents the *cost* and *benefits* associated with a set of decisions about the structure of a software project. Note the large variance in the possible cost and benefits from the different possible decisions.

Faced with such large data clouds, it is hard to demonstrate that any particular decision leads to a particular outcome. What is required is some method for condensing these clouds of uncertainty without expensive data collection for all the uncertain parameters. Ideally, condensation methods should be *minimal*; i.e. they require a commitment to only a small portion of the uncertain variables within a model.

This papers experiments with minimal condensation using the TAR2 *treatment learner* [19,24,33-36] A treatment learner seeks the *least number* of attribute ranges that *most differentiate* between desired and undesired behavior. Figure 2 shows how TAR2 can be applied incrementally to explore data clouds. A simulator executes a model generated by some manual modelling process. TAR2 reduces the data generated by the simulator to a set of *proposed treatments*. After some discussion, users add the *approved treatments* to a growing set of constraints for the simulation. The cycle repeats until users see no further improvement in the behavior of the simulator.

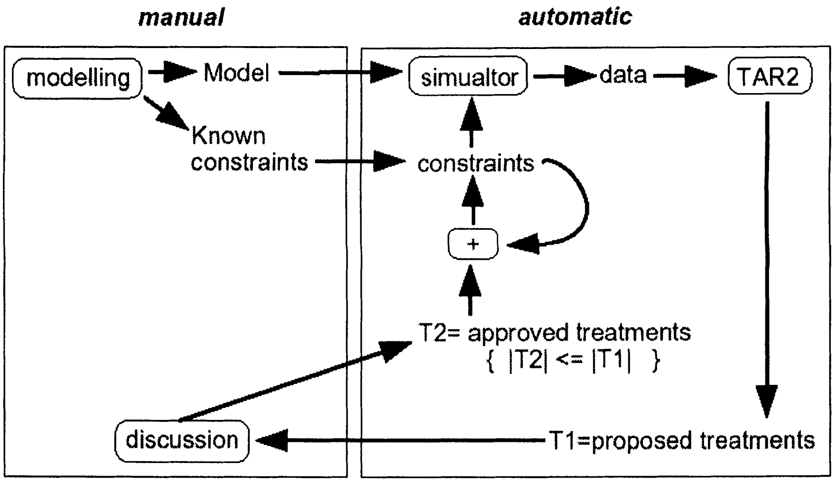


Figure 2. Incremental treatment learning.

Experiments with this approach have shown that TAR2 can:

- *Reduce the variance* of values within a data cloud
- *Improve the mean* of values within data clouds

For example, Figure 1.iii and Figure 1.iv show the results of applying incremental treatment learning to Figure 1.i and Figure 1.ii. Note that in both studies, the mean of the benefits increased, the mean of the costs decreased, and the variance in both measures was greatly decreased.

The notion that extra constraints can reduce the space of uncertainties is hardly surprising. However, what is surprising is *how few* extra constraints TAR2 needs to condense (e.g.) Figure 1.ii to Figure 1.iv and how *easily* TAR2 can automatically find those constraints. The claim of this paper is that:

> In the average case, a simple algorithm (TAR2) can quickly find a very small number of key constraints that result in massive condensations of data clouds towards some desired goal.

There are three implications of this claim. Firstly, even when we don't know exactly what is going on within a model, it is possible to define minimal strategies to grossly decrease the uncertainty in that model's behavior.

Secondly, even if we aren't sure about the impact of certain decisions, we can be sure that certain other decisions will be ineffective. Decisions about treatment variables will override decisions about variables not found within a treatment. Hence, decisions about variables outside the treatments are redundant.

Thirdly, incremental treatment learning can reduce the cost of software modeling. Before applying elaborate modeling techniques or tools, it is wise to try cheaper and simpler techniques. Our results here show that even hastily built incomplete models can be used for effective decision making. Since much can be learnt, even from sketchy data, it may be possible to avoid elaborate and extensive and expensive metrics collection. Further, once the treatments are known, then a minimal metrics collection program can be defined, just for the few variables in the treatments.

The rest of this paper describes the details of our condensation technique. TAR2 was motivated by *funnel theory* which is a claim that most decisions are redundant or irrelevant. In models containing funnels, a small number of key variables are enough to control a model, despite the large range of possibilities outside the funnel. Funnel theory is discussed in §2. Our algorithm for finding the key decisions within the funnels is discussed in §3. Case studies are then explored in §4 where TAR2 can reduce the variance and improve the mean of three case studies. After that, §5 discusses when this approach may not be appropriate and §6 discusses related work.

2. FUNNEL THEORY

The premise of this paper is that within the space of possible decisions, there exist a small number of key decisions that determine all others. After Menzies, Easterbrook, Nuseibeh, and Waugh, we call this premise *funnel theory*- the metaphor being that all processing runs down the same narrow funnel [32].

To introduce funnels, we first say that a decision space supports *reasons*; i.e. chains of reasoning that link inputs in a certain context to desired goals. Chains have links of at least two types. Firstly, there are links that clash with other links. Secondly, there are the links that depend on other links. One method of optimizing the decision making process would be to first decide about the non-dependent clashing links. These are the *key decisions* since they determine most of the other non-key decisions.

For example, suppose the following decision space is explored using the invariant `no_good(X, ¬X)` and everything that is not a context or a goal is open to debate:

```
                   a →  b →  c  →  d →  e
     context1  → f →  g →  h  →  I →  j →  goal
     context2  → k →  g →  l  →  m →  j →  goal
                   n →  o →  p  →  q →  e
```

Like any model, any of `(a, b, ..q)` is subject to discussion. However, in the context of reaching some specified goals from *context1* and *context2*, the only important discussions are the clashes `(g, ¬g, j, ¬j)` (the `(e, ¬e)` clash is not exercised in the context of context1; context2 ⊢ goal, since no reason uses `e` or `¬e`). Further, since `(j, ¬j)` are fully dependent on `(g, ¬g)`, then the core decision must be about variable (`g`) with two disputed values: true and false.

The *funnel* of a decision space contains the non-dependent clashing links; e.g. {g}. The decisions with *greatest information content* are the decisions about the funnel variables, since these variables set the others. If the space contains *narrow funnels* (i.e. funnels with small cardinality) then the total decision space can be greatly reduced to a small number of highly informative disputes about funnel variables. Analysts are still free to debate whatever they want (and they will, seemingly endlessly), but with this approach, a funnel-aware analyst can steer the discussion towards the issues that tell us most about a domain. The net effect can be less argument. Suppose our analysts agree that `g` is true, then in the context of arguing about how `context1; context2 ⊢ goal`, the decision space reduces to:

```
     context1  → f →  g →  h  →  I →   j →  goal.
```

The reasoning starting with `k` has been culled since, by endorsing g, we must reject all lines of reasoning that use `¬g`. In addition, the reasoning starting with a, n are ignored since they are irrelevant in this context; i.e. they do not participate in reaching a desired goal. Further, in this context, there is little point arguing about `(f,h,i,j)` since if any of these are false, then no goal can be reached.

This small example suggests how funnels can condense data clouds. Data clouds are the result of a wide variation in model behavior. Such variation comes from choices within a model relating to uncertain ranges. The more commitments we make about funnel variables, the more we collapse the space of possibilities outside the funnel. Hence, decisions about funnel variables condense data clouds, since they restrict the behavior of a system. Decision making in spaces containing funnels can be simple and short. Once values for the funnel variables are decided, all other decisions become redundant. In the above example, we have a decision space containing potentially 2^{17} = 131,072 debates about 16 Boolean variables `{a,b,..q}`. A decision about one variable (i.e. "is `g` true or false?") has reduced this space to one option.

Relying on narrow funnels may seem an overly optimistic approach. Yet a literature review suggests that such optimism is well-founded. There are many examples of funnel-like behavior in the literature. For example Horgan & Mathur [23] report that testing often exhibits a *saturation* effect; i.e. most program paths get exercised early with little further improvement as testing continues. Saturation is consistent with funnels controlling the reachable parts of a program. If these funnels were narrow, there would be few options in a program's execution and test inputs would quickly sample them all. Further testing over systems with narrow funnels would yield little further information since anything not connected to the funnels would be, by definition, unreachable.

An analogous effect to saturation is *homogenous propagation* since in the program mutation literature (a *program mutant* is a syntactically valid but randomly selected variation to a program; e.g. swapping all plus signs to a minus sign). Despite numerous perturbations on data values using a program mutator, Michael found that in 80 to 90% of cases, there were no changes in the behavior of a range of programs [40]. Another study compared results using X% of a library of mutators, randomly selected ($X \in \{10,15,...40,100\}$). Most of what could be learnt from the program could be learnt using only X=10% of the mutators; i.e. after a very small number of mutators, new mutators acted in the same manner as previously used mutators [54]. The same observation has been made elsewhere in the mutation literature [1,8]. Like saturation, homogenous propagation is consistent with funnel theory. If the overall behavior of a system is determined by a small number of key variables, then random mutation is unlikely to find those variables and the net effect of those mutations would be very small.

Homogenous propagation is observed in procedural programs. An analogous effect has been seen in declarative systems; i.e. most choices within a declarative set of constraints have little effect on the average behavior. Menzies & Waugh studied choices in millions of mutations of a nondeterministic system. In their *abductive framework* [27,31], a consistent set of choices generated a *world* of belief. Given N binary choices, there are theoretically 2^N possible worlds. However, after studying millions of generated worlds they found the maximum number of goals found in any world was often close to the number of goals found in a world selected at random [39] (on average, the difference was less than 6%). This observation is inexplicable without narrow funnels. If choices had a large impact on what was reached within a declarative system, then there should be much variability in what is found in each world. Since the observed variability was so small, the number of critical choices (a.k.a. funnel variables) must also be small.

In fact, the concept of a funnel has been reported in many domains under a variety of names including:

- *Master-variables* in scheduling [15];
- *Prime-implicants* in model-based diagnosis [47] or machine learning [46], or fault-tree analysis [29].
- *Backbones* in satisfiability [41, 51];
- *The dominance filtering* used in Pareto optimization of designs [26];
- *Minimal environments* in the ATMS [16];
- The *base controversial assumptions* of HT4 [31].

Whatever the name, the core intuition in all these terms is the same: what happens in the total space of a system can be controlled by a small critical region. The frequency of the funnel effect have made Menzies & Singh suspect that funnels are some average case phenomenon that is emergent in decision spaces [37]. To test this, they consider a device that can choose between a narrower and a wider funnel. Let some goal in a system be reachable by a narrow funnel M or a wide funnel N shown in Figure 3.

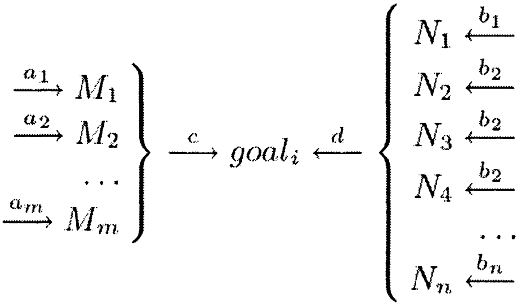


Figure 3. Alternate funnels that lead to some goal.

Under what circumstances will the narrow funnel be favored over the wide funnel? The following definitions let us answer this question:

- Let the cardinality of the narrow funnel and wide funnels be m and n respectively.
- Each m members of the narrow funnel are reached via a path with probability ai while each n members of the wider funnel are reached via a path with probability bi.

- Two paths exist from the funnels to this goal: one from the narrow neck with probability c and one from the wide neck with probability d. Therefore, the probability of reaching the goal via the narrow pathway is

$$narrow = c\prod a_i$$

while the probability of reaching the goal via the wide pathway is

$$wide = d\prod b_i$$

With these definitions, the Menzies & Singh study can be redefined as the search for conditions under which

$$(narrow / wide = R) > t \qquad (1)$$

where t is some threshold value. To explore Equation 1, Menzies & Singh built a small simulator of Figure 3, and performed 150,000 runs using different distributions for a_i; b_i; c; d and a wide range of values for m; n. The results are shown in Figure 4. For comparison purposes, the size of the two funnels is expresses as a ratio `alpha` where `n=alpha*m`.

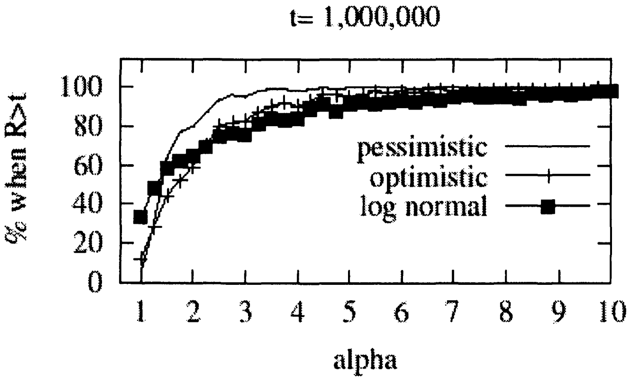


Figure 4. 10,000 runs of the funnel simulator. Y-axis shows what percentage of the runs satisfiess (narrow/wide=R)>t . The pessimistic, lognormal, and optimistic distributions assume a worst-case, average-case, and best-case (respectively) distribution for {ai, bi, ci, di}. For more details, see[37].

As might be expected, at `alpha=1` the funnels are the same size and the odds of using one of them is 50%. As alpha increases, then increasingly `R>t` is satisfied and the narrower funnel is preferred to the wider funnel. The effect is quite pronounced. For example, for all the studied distributions, after the wider funnel is 2.25 times bigger than the narrow funnel, then in 75% or more of the random searches, accessing the narrow funnel is at least 1,000,000 times more likely as accessing the wider funnel (see the lower graph of Figure 4). Interestingly, as the probability of using any of a_i, b_i, c_i, d_i decreases, the odds of using the narrow funnel increase (see the *pessimistic curves* in Figure 4). That is, narrow funnels are likely, especially in spaces that are difficult to search.

The average case analytical result of Menzies & Singh is suggestive evidence, but not conclusive evidence, that narrow funnels are common. Perhaps a more satisfying test for narrow funnels would be to check if, in a range of applications, a small number of variables are enough to control the other variables in a model. The rest of this paper implements that check.

3. FINDING THE FUNNEL

A traditional approach to funnel-based reasoning is to find the funnels using some dependency-directed backtracking tool such as the ATMS [16] or HT4 [31]. Dependency-directed backtracking is very slow, both theoretically and in practice [31]. Further, in the presence of narrow funnels, it may be unnecessary. There is no need to *search* for the funnel in order to *exploit* it. Any reasoning pathway to goals must pass through the funnels (by definition). Hence, all that is required is to find attribute ranges that are associated with desired behavior.

outlook	temp(°F)	humidity	windy?	class
sunny	85	86	false	none
sunny	80	90	true	none
sunny	72	95	false	none
rain	65	70	true	none
rain	71	96	true	none
rain	70	96	false	some
rain	68	80	false	some
rain	75	80	false	some
sunny	69	70	false	lots
sunny	75	70	true	lots
overcast	83	88	false	lots
overcast	64	65	true	lots
overcast	72	90	true	lots
overcast	81	75	false	lots

Figure 5. A log of some golf playing behavior.

TAR2 is a machine learning method for finding attribute ranges associated with desired behavior. Traditional machine learners generate classifiers that assign a class symbol to an example [44]. TAR2 finds the difference between classes. Formally, the algorithm is a *contrast set learner* [4] that uses *weighted classes* [9] to steer the inference towards the preferred behavior. The algorithm differs from other learners in that it seeks contrast sets of *minimal* size.

TAR2 can best be introduced via example. Consider the log of golf playing behavior shown in Figure 5. This log contains four attributes and 3 classes. Recall that TAR2 accesses a *score* for each class. For a golfer, the classes in Figure 5 could be scored as *none=2* (i.e. worst), *some=4*, *lots=8* (i.e. best).

TAR2 seeks attribute ranges that occur more frequently in the highly scored classes than in the lower scored classes. Let `a:r` be some attribute range e.g. *outlook.overcast*. Δ`a:r` is a heuristic measure of the worth of a:r to improve the frequency of the best class. ∇`a:r` uses the following definitions:

`X(a:r)` the number of occurrences of that attribute range in class X; e.g. lots(outlook.overcast)=4.

`all(a:r)` total number of occurrences of that attribute range in all classes; e.g. all(outlook.overcast)=4.

`best` the highest scoring class; e.g. `best` = lots;

`rest` the non-best class; e.g. `rest` = {none; some};

`$Class` score of a class `Class` is `$Class`.

Δ`a:r` is calculated as follows:

```
Δa:r=(for X∈rest do
                Δa:r := Δa:r*(($best- $X) * (best(a:r) -
X(a:r)))
       )/all(a:r)
```

For example, when `a.r` is *outlook.overcast*, then Δoutlook:overcast is

```
(((8-2)*(4-0))+((8-4)*(4-0)))/(4+0+0)=40/4=10
```

The attribute ranges in our golf example generate the Δ histogram shown in Figure 6. Note that *outlook=overcast*'s Δ is the highest, potentially most effective, attribute range.

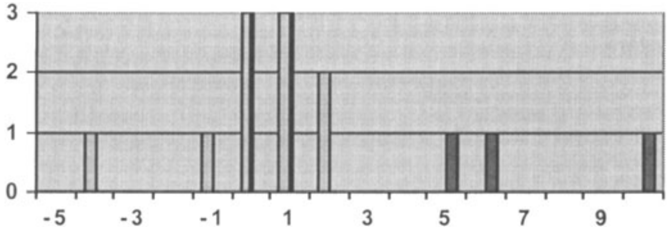


Figure 6. Δ distribution seen in golf data sets. The X-axis shows the range of Δ values seen in the gold data set. The Y-axis shows the number of attribute ranges that have a particular Δ.

A *treatment* is a subset of the attribute ranges with an *outstanding* Δa=r value. For our golf example, such attributes can be seen in Figure 6: they are the outliers with outstandingly large Δs on the right-hand-side. (These outliers include *outlook=overcast*).

To *apply* a treatment, TAR2 rejects all example entries that contradict the conjunction of attribute ranges in the treatment. The ratio of classes in the remaining examples is compared to the ratio of classes in the original example set. The *best treatment* is the one that most increases the relative percentage of preferred classes. In the case where N treatments increase the relative score by the same amount, then N *best treatments* are generated and TAR2 picks one at random. In our golf example, a single best treatment was generated containing *outlook=overcast*; Figure 7 shows the class distribution before and after that treatment, i.e. if we choose a vacation location that is generally overcast, then in 100% of cases we should be playing lots of golf, all the time.

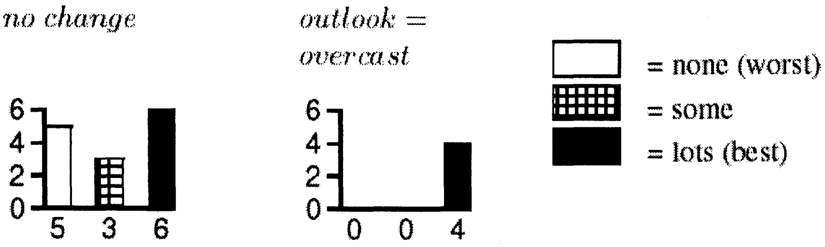


Figure 7. Finding treatments that can improve golf playing behavior. With no treatments, we only play golf lots of times in 6/(5+3+6) = 57% of cases. With the restriction that outlook=overcast, then we play golf lots of times in 100% of cases.

Claim	Notes
c1	among the few vital goals
c2	a claim by David Parnasî[42]
c3	few assumptions among interacting modules
c4	expected size of data is huge
c5	many implementors familiar with ADT (from domain experts)

Figure 8a. the claims of Fig 8b.

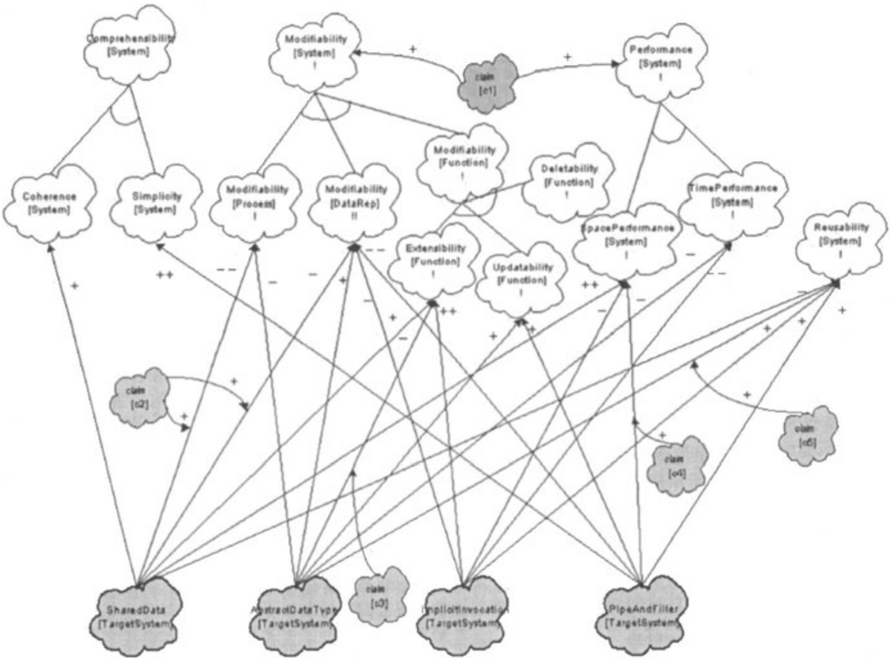

Comprehensibility [System]
Modifiability [System] !
claim [c1]
Performance [System] !
Modifiability [Function] !
Deletability [Function] !
Coherence [System]
Simplicity [System]
Modifiability [Process] !
Modifiability [DataRep] !!
Extensibility [Function] !
Updatability [Function] !
SpacePerformance [System] !
TimePerformance [System] !
Reusability [System] !
claim [c2]
claim [c4]
claim [c5]
claim [c3]
SharedData [TargetSystem]
AbstractDataType [TargetSystem]
ImplicitInvocation [TargetSystem]
PipeAndFilter [TargetSystem]

Figure 8b. A model that assesses architectural choices within software. Options within the model are the leaf nodes shown in gray. These options can be architectural decisions such as the use of abstract data types, implicit invocation, pipe & flter methods, or shared data. Some links in the model are dependant of various claims c1,... , c5 shown in Fig 8a. For example, claim[c2] is Parnas's [42] argument that having a single share data model across an entire application has a negative impact on the modifiability of that process. The inference rules of this diagram are shown in Fig 8c.

The benefit of the this network is the benefit computed for the top-level node good. This benefit is defined recursively as follows:

- The benefit of a leaf node is 1 if it is selected, or 0 otherwise. Leaf nodes represent choices in the network. Leaf nodes are shown in gray in **Fig 8b.**
- The benefit of a non-leaf node is computed from its input influences.
- An influuence of an edge on an upstream node is the product of the edge weight and the benefit of the downstream node.
- Edge weights are set by tables that offer numeric values for (++,+,=,-,--).
- Nodes are either disjunctions or conjunctions. Conjunctions are shown as diamonds in **Fig 8b.**. The benefit of a conjunction is minimum of the input influences. The benefit of a disjunction is the average of the input influences. For a rationale on why these rules were selected, see [10]. In summary: these rules were not unreasonable and the users wanted it that way. Future experiments in this domain will explore variants of these rules.

Figure 8c. inference rules for Fig 8b.

4. CASE STUDIES

This section presents three examples of incremental treatment learning. The examples are sorted by model size: smallest to largest. The largest and final model is too detailed to explain here but the second largest model is explained in sufficient detail for the reader to reproduce the entire experiment. In all examples, the objective of incremental treatment learning is to find a subset of all possible decisions that *reduces the variance* and *improves the mean* of the important variables within a data cloud.

4.1. Case Study A: Software Architectures

Figure 8 shows some architectural assessment knowledge taken from Shaw & Garlan's Software Architectures book [49]. The knowledge is expressed in our variant of the *softgoal* notation of Chung et al. [12]. In the softgoal approach, a *softgoal* is distinguished from a normal goal as follows:

- While a goal has well-defined non-optional feature of a system that *must* be available, a softgoal is a goal that has no clear cut criteria for success.
- While goals can be conclusively demonstrated to be *satisfied* or *not satisfied*, softgoals can only be *satisfied* to some degree.

Much is under-constrained in Figure 8. In fact, there are $4^{21} * 2^{9} \approx 10^{15}$ possibilities within this model:

- The nine boolean choices in the model are leaf nodes representing software architecture options or claims about the application. Hence, there are 2^{9} combinations of these choices.
- Edges between nodes in Figure 8 are annotated with a symbol denoting how strongly the downstream node impacts the upstream node. These

annotations are {++,+,=,-,--} denoting *makes; helps; equals; hurts; breaks*(respectively). For the sake of exposition, we say that the values for four of these annotations come from a range of 21 possible values:

$$1 \geq X_{makes} > X_{helps} > X_{hurts} > X_{breaks} > -1$$

$$X_I \in \{-1, -0.9, -0.8; \ldots 0, \ldots 0.9, 1\} \quad (2)$$

(The exact value of equals is not varied since this annotation is used to propagate influences unchanged over an edge; i.e. *weight (equals) =1*). Hence, in the worst case, there are $4^{21} \approx 10^{12}$ possible edge weights.

These possibilities generate a wide range of behavior. Our softgoal interpreter [10] computes a cost and benefit figure resulting from a selection of edge weights and choices in diagrams like Figure 8 (the details of this computation are discussed in the *inference rules* table of Figure 8). Figure 1.i shows the range of benefits and costs seen after 10,000 random selection of choices and edge weights. Note the large variance in these figures.

To apply incremental treatment learning for this case study, we first require a scoring scheme for the different classes. In 10,000 runs of Figure 8, with no constraints on any selections, the observed costs ranged from 1 to 4 and the benefit ranged from -18 to 12 (see Figure 1.i). Since high benefit and low cost is preferable to high cost and low benefit, these ranges were scored as shown in Figure 9. In that figure, the best range is benefit ≥ 12 and cost = 1 and the worst range is benefit ≤18 and cost = 4.

scoring function:

	Cost			
Benefit	1	2	3	4
12	1	2	3	4
6	5	6	7	8
0	9	10	11	12
-6	13	14	15	16
-12	17	18	19	20
-18	21	22	23	24

Figure 9. Class scoring function

TAR2 was applied to Figure 8 four times. Each round comprised 10,000 runs where:

- Edge weights were selected at random at the start of each run from Equation 2.
- From the space of remaining choices, architectural options and claims were selected at random.

Initially, no restrictions were imposed on the architectural options and claims. This generated the ranges of cost and benefit shown in Figure 1.i.

Such a data cloud is hard to read. A more informative representation is the *percentile matrix* of Figure 10. Each cell of this matrix shows the percent of runs that falls into a certain range. Each cell is colored on a scale ranging from white (0%) to black (100%).

	Cost				
Benefit	1	2	3	4	Totals
12					
6		1	2	1	4
0	13	19	15	4	51
-6	10	12	4	1	27
-12	4	6	2		12
-18	3	2	1	1	7
Totals	30	40	24	6	100

Figure 10. Percentage distributions of benefits and costs seen in 10,000 runs of Figure 8, assuming Equation 2 and a random selection of architectural options and claims.

round 1

	Cost				
Benefit	1	2	3	4	Totals
12		1	2	1	4
6		5	9	3	17
0	11	30	26	7	74
-6	1	2	1	1	5
-12					
-18					
Totals	12	38	38	12	100

$KEY1 = (\mathit{claim}[c1] = \mathit{yes}) \wedge (\mathit{pipe} \,\&\, \mathit{filter}[t \arg \mathit{etsystem}] = \mathit{yes})$

round 2:

	Cost				
Benefit	1	2	3	4	Totals
12		3	6		9
6	4	17	14		35
0	28	28			56
-6					
-12					
-18					
Totals	32	48	20		100

$KEY_2 = KEY_1 \wedge (\mathit{shareddata}[t \arg \mathit{etsystem}] = \mathit{yes}) \wedge$
$(\mathit{implicitinvocation}[t \arg \mathit{etsystem}] = \mathit{no})$

round 3:

	Cost				
Benefit	1	2	3	4	Totals
12		12			12
6		39			39
0		48			48
-6		1			1
-12					
-18					
Totals		100			100

$KEY3 = KEY_2 \wedge (\mathit{abstractdatatype}[t \arg \mathit{etsystem}] = \mathit{no})$
$\wedge (\mathit{claim}[c3] = \mathit{no})$

round 4

	Cost				
Benefit	1	2	3	4	Totals
12		20			20
6		38			38
0		42			42
-6					
-12					
-18					
Totals		100			100

$KEY_4 = KEY_3 \wedge$
$(\mathit{claim}[c2] = \mathit{yes}) \wedge (\mathit{claim}[c4] = \mathit{yes})$

Figure 11. Percentile matrices showing four rounds of incremental treatment learning for Fig 8.

Figure 11 shows the results of applying incremental treatment learning to Figure 8. Each round took the key decisions learnt by TAR2 from 10,000 examples generated in the previous round. 10,000 more runs were then performed, with the selection of architectural options and claims restricted according to the current set of key decisions. Note that as the key decisions accumulate the variance in the behavior decreases and the means improve (decreased cost and increased benefit).

This experiment stopped after four rounds since there was little observed improvement between round 3 and round 4. Figure 1.iii shows the results of the round 3, not round 4; i.e. this experiment returned the results from round 3, and not round 4. By stopping at round 3, analysts can avoid excessive decision making since they need never discuss c2; c4; c5 with their users. Alternatively, if in some dispute situation, an analyst could use c2; c4; c5 as bargaining chips. Since these claims have little overall impact, our analyst could offer them in any configuration as part of some compromise deal in exchange for the other key decisions being endorsed.

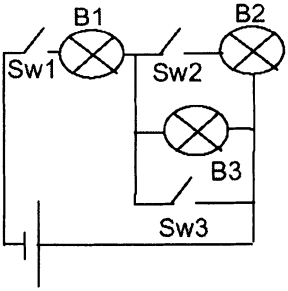


Figure 12. A qualitative circut. from [5].

4.2. Case Study B: Circuit Design

Our next example contains a model somewhat more complex than §4.1. This example is based on models first developed by Bratko to demonstrate principles of qualitative reasoning [5].

While our last example generated cost and benefit figures for a software project, this example is a qualitative model of a circuit design shown in Figure 12. Such qualitative descriptions of a planned piece of software might appear early in the software design process. We will assume that the goal of this circuit is to illuminate some area; i.e. *the more bulbs that glow, the better*.

For exposition purposes, we assume that much is unknown about our circuit. All we will assume is that the topology of the circuit is known, plus some general knowledge about electrical devices (e.g. the voltage across components in series is the sum of the voltage drop across each component). What we don't know about this circuit are the precise quantitative values describing each component.

```
% sum(X,Y,Z).
 sum(+,+,+).    sum(+,0,+).   sum(+,+,Any).
 sum(0,+,+).    sum(0,0,0).    sum(0,-,-).
sum(-,+,Any).   sum(-,0,-).    sum(-,-,-).
```

Figure 13. Qualitative mathematics using a Prolog syntax[5].

```
% bulb(Mode,   Light, Volts, Amps)
  bulb(blown,  dark,  Any,   0).
  bulb(ok,     light, +,     +).
  bulb(ok,     light, -,     -).
  bulb(ok,     dark,  0,     0).

% num(Light,Glow). %
switch(State,Volts,Amps)
  num(dark, 0).      switch(on,   0,
Any).
  num(light,1).      switch(off,  Any,
0).
```

Figure 14. Definitions of qualitative bulbs and switches. Adapted from [5].

When quantitative knowledge is unavailable, we can use qualitative models. A qualitative model is a quantitative mode whose numeric values x are replaced by a qualitative value x' having one of three qualitative states: +, -, 0; i.e.

```
x' = + if x > 0
x' = 0 if x = 0
x' = - if x < 0
```

The sum relation of Figure 13 describes our qualitative knowledge of addition using a Prolog notation. In Prolog, variables start with upper case letters and constants start with lower-case letters or symbols. For example,

```
sum(+, +, +)
```

says that the addition of two positive values is a positive value. There is much uncertainty within qualitative arithmetic. For example,

```
sum(+,-,Any)
```

says that we cannot be sure what happens when we add a positive and a negative number. The bulb relation of Figure 14 describes our qualitative knowledge of bulb behavior. For example,

```
bulb(blown,dark,Any,0)
```

says that a blown bulb is dark, has zero current across it, and can have any voltage at all. Also shown in Figure 14 are the num and switch relations. Num defines how bright a dark or light bulb glows while switch describes our qualitative knowledge of electrical switches. For example,

```
switch(on, 0, Any)
```

says that if a switch is on, there is zero voltage drop across it while any current can flow through it.

```
circult(switch(Sw1, VSw1, C1),
              bulb(B1, L1, VB1, C1),
              switch(Sw2, VSw2, C2),
              bulb(B2, L2, VB2, C2),
              switch(Sw3, VSw3, CSw3),
              bulb(B3, L3, VB3, CB3),
              Glow) :-
        VSw3 = VB3,
        sum(VSw1, VB1, V1),          %      9 options
        sum(V1, VB3, +),             %      1 option
        sum(VSw2, VB2, VB3),         %      9 options
        switch(Sw1, VSw1, C1),       %      2 options
        bulb(B1, L1, VB1, C1),       %      4 options
        switch(Sw2, VSw2, C2),       %      2 options
        bulb(B2, L2, VB2, C2),       %      4 options
        switch(Sw3, VSw3, CSw3),     %      2 options
        bulb(B3, L3, VB3, CB3),      %      4 options
        sum(CSw3, CB3, C3),          %      9 options
        sum(C2, C3, C1),             %      9 options
        num(L1, N1),
        num(L2, N2),
        num(L3, N3),
        Glow is N1+N2+N3.
```

Figure 15. Figure 12 modeled in Prolog. Adapted from [5].

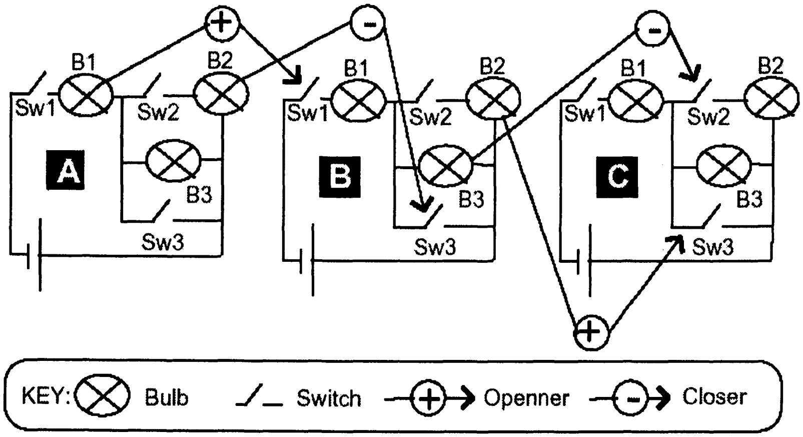


Figure 16. A device modeled using the Prolog of Figure 15.

The circuit relation of Figure 15 describes qualitative knowledge of a circuit using bulb, num, sum and switch. This relation just records what we know of circuits wired together in series and in parallel. For example:

- Switch 3 and Bulb3 are wired in parallel. Hence, the voltage drop across these components must be the same (see line 8)
- Switch 2 and Bulb2 are wired in series so the voltage drop across these two devices is the sum of the voltage drop across each device. Further, this summed voltage drop must be that same as the voltage drop across the parallel component Bulb3 (see line 11).
- Switch1 and Bulb1 are in series so the same current C1 must flow through both (see line 12 and line 13).

In order to stress test our method, our case study will wire up three copies of Figure 15 in such a way that solutions to one copy won't necessarily work in the other copies. Figure 16 shows our circuit connected by a set of *openers* and *closers* that open/close switches based on how much certain bulbs are glowing. For example, the *closer* between bulb *B2A*and switch *Sw1B* means that if *B2A* glows then *Sw1B* will be closed. These openers and closers are defined in Figure 17. The full model is shown in Figure 18.

```
% inf(Sign,Bulb,Switch)
inf(Inf,bulb(_,Shine,_,_),switch(Pos,_,_)):-
  infl(Inf,Shine,Pos).

% infl(Sign,Glow,SwitchPos)
  infl(+,dark, off). infl(+,light, on).
  infl(-,dark, on). infl(-,light, off).
```

Figure 17. The infl/3 predicate used to connect bulb brightness to switches.

The less that is known about a model, the greater the number of possible behaviors. This effect can easily be seen in our qualitative model. Each line of Figure 15 is labeled with the number of possibilities it condones: i.e.

```
9*1*9*2*4*2 * 4 * 2 * 4 * 9 * 9 = 3,359,232
```

Copied three times, this implies a space of up to $3{,}359{,}232^3 = 10^{19}$ options. Even when many of these possibilities are ruled out by inter-component constraints, the circuits relation of Figure 18 can still succeed 5,228 times (some sample output is shown in Figure 19).

Given the goal that the *more* lights that shine, the better the circuit, we assume 10 classes: 0,1,2,3,...9, one for every possible number of glowing bulbs. As shown in Figure 20, within the 35,228 runs, there are very few lights shining. In fact, on average within those runs, only two lights are shining. TAR2's mission is to explore the space, trying to find key decisions which, when applied to the circuit, can most improve this low level of lighting.

4.2.1. Round 1

After learning treatments from the all 35,228 initial runs, and applying them to the data, TAR2 generated Figure 21. In summary, Figure 21 is saying that making a single decision will change the average illumination of the circuit from 2 to 5 (if Sw2C=off) or 6 (if Sw3C=on).

```
circuits :-
       % some initial conditions
       value(light,bulb,B1a),
       % Uncomment to constrain Sw2c
       % value(off,switch,Sw2c),
       % Uncomment to constrain Sw1c
       % value(on,switch,Sw1c),
       % Uncomment to constrain Sw3c
       % value(on,switch,Sw3c),
             % explore circuit A
       circuit(Sw1a,B1a,Sw2a,B2a,Sw3a,B3a,GlowA),
             % let circuit A influence circuit B
             inf(+,B1a,Sw1b),
             inf(-,B2a,Sw3b),
             % let circuit B influence circuit C
       circuit(Sw1b,B1b,Sw2b,B2b,Sw3b,B3b,GlowB),
             % propagate circuit B to circuit C
             inf(-,B3b,Sw2c),
             inf(+,B2b,Sw3c),
             % explore circuit C
       circuit(Sw1c,B1c,Sw2c,B2c,Sw3c,B3c,GlowC),
             % compute total shine
             Shine is GlowA+GlowB+GlowC,
         % make one line of the examples
     format('~p,~p,~p,~p,~p,~p,~p,~p,~p',
       [Sw1a,Sw2a,Sw3a,Sw1b,Sw2b,Sw3b,
        Sw1c,Sw2c,Sw3c]),
         format('~%,~%,~%,~%,~%,~%,~%,~%,~%,~p
                [B1a,B2a,B3a,B1b,B2b,B3b
                 ,B1c,B2c,B3c,Shine]),nl.

       data :- tell('circ.data'),
             forall(circuits,true), told.

       % some support code
       value(Sw,    switch, switch(Sw,_,_)).
       value(Light, bulb,bulb(_,Light,_,_)).

       :- format_predicate('%',bulbIs(_,_)).

       bulbIs(_,bulb(X,_,_,_)) :-
             var(X) -> write('?') |write(X).
       portray(X) :- value(Y,_,X), write(Y).
```

Figure 18. Fig 16 expressed in Prolog.

Swla	Sw2a	Sw3a	Swlb	...	B3b	Blc	B2c	B3c	Shine
on	off	off	on	...	blown	blown	blown	blown	2
on	off	off	on	...	blown	blown	blown	blown	2
on	off	off	on	...	blown	blown	blown	blown	2
on	off	off	on	`...	blown	ok	blown	blown	2
on	off	off	on	...	ok	blown	blown	blown	2
on	off	off	on	...	blown	blown	blown	ok	2
on	off	off	on	...	ok	blown	ok	blown	3
on	off	off	on	...	ok	blown	ok	ok	3
on	off	off	on	...	ok	blown	blown	blown	3
on	off	off	on	...	blown	ok	ok	blown	5

Figure. 19. Some output seen in circ.data generated using data (line 32 of Figure 18). Columns denote values from Figure 16. For example, Swla and Swlb denotes switch 1 in ciruit A and ciruit A respectively.

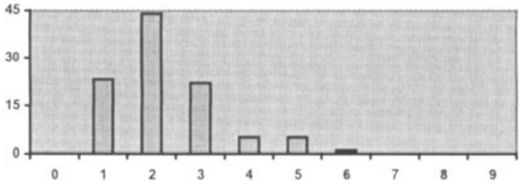


Figure 20. Frequency count of number of bulbs glowing in the 35,228 solutions of circuits of Figure 18.

For exposition purposes, this example assumed that something prevents our users from making this key decision; i.e. Sw3C=on. Our experience with incremental treatment learning is that this is often the case. When users are presented with the next key decision, they often recall some key knowledge that they neglected to mention previously. In this case, we assumed that it is preferable if switch 3 in circuit C is not closed- since that would violate (say) the warranty on circuit C. Our analysts therefore agreed to the next best treatment, i.e. Sw2C=off; shown in Figure 21, left hand side (LHS).

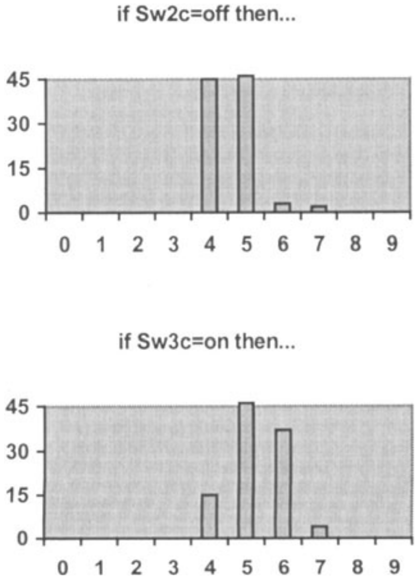


Figure 21. Run#1 of TAR2 over the data seen in Figure 20.

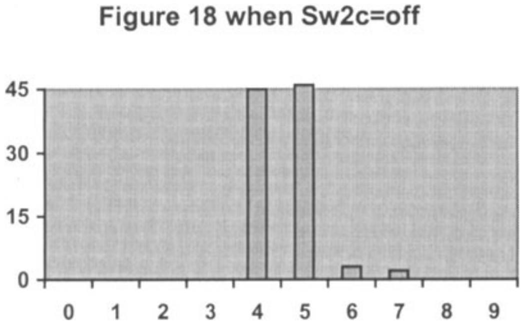


Figure 22. Frequency count of number of bulbs glowing in the 3,264 solutions of `circuits` of Figure 18 when Sw2C=off.

4.2.2. Round 2

After constraining the model to Sw2C=off (i.e. by uncommenting line 5 in Figure 18), fewer behaviors were generated: 3,264 as compared to the 35,228 solutions seen previously. The frequency distribution of the shining lights in this new situation is shown in Figure 22.

Happily, Figure 22 has the same distribution as Figure 21.LHS; i.e. in this case, when the constraints proposed by TAR2's best treatment were applied to the model, the resulting new behavior of the model matched the new behavior predicted by the treatment.

Executing TAR2 again found the next most informative decision, as shown in Figure 23. Here, TAR2 said that our best treatment would be to guarantee that bulb 3 in circuit C is never blown. Perhaps this is possible if we were to use better light bulbs with extra long life filaments. However, for the sake of exposition, we assumed that there is no budget for such expensive hardware. Hence, to avoid this expense, our analysts agreed that always closing switch 1 in circuit C (as proposed by Figure 23.LHS) is an acceptable action.

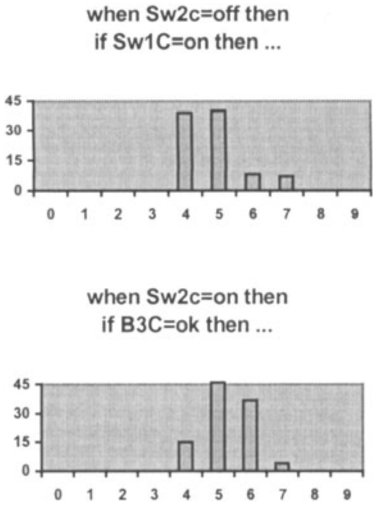


Figure 23. Run #2 of TAR2 over the data seen in Figure 22.

4.2.3. Round 3

After constraining the model to Sw1C=on (i.e. by uncommenting line 7 in Figure 18), fewer behaviors were generated: 648 as compared to the 3,264 solutions seen previously. The frequency distribution of the shining lights in this new situation is shown in Figure 24.

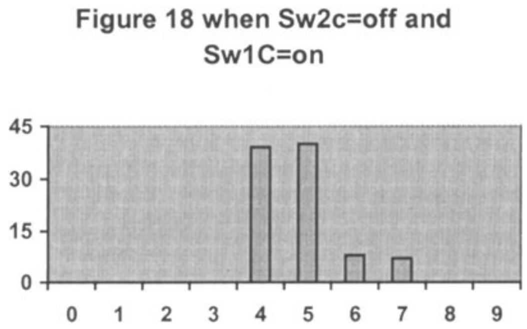


Figure 24. Frequency count of number of bulbs glowing in the 648 solutions of `circuits` of Figure 18 when Sw2C=off and Sw1C=on.

Figure 24 has the same distribution as Figure 23.LHS. That is, once again, TAR2's predictions proved accurate. Executing TAR2 again generated Figure 25 and finds the next most informative decision.

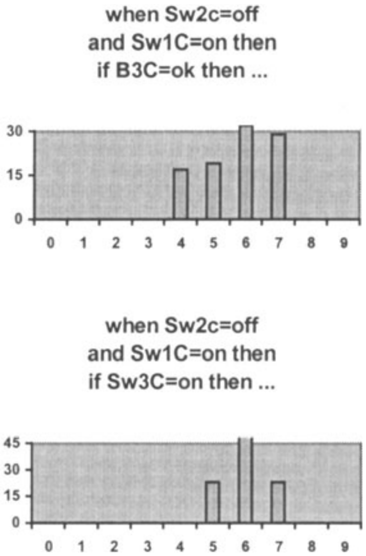


Figure 25. Run#3 of TAR2 over the data seen in Figure 24.

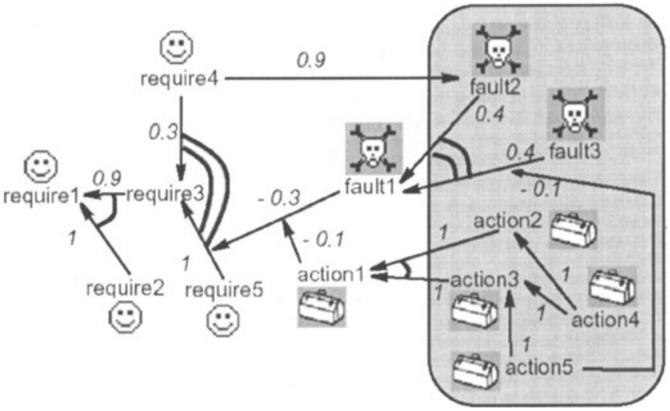
The cycle could stop here since the next best treatments are not acceptable. Figure 25.LHS wants to use overly expensive hardware to ensure that bulb 3 in circuit C is always not blown. Figure 25.RHS wants to use an undesirable action and close switch 3 in circuit C. However, our engineers have enough information to propose some options to their manager: if they increase their hardware budget, they could make the improvements shown in Figure 25.LHS. Alternatively, if there was some way to renegotiate the warranty, then the improvements shown in Figure 25.RHS could be achieved.

To verify this, our engineers continue constraining Figure 18 to the case of Sw3c=on by uncommenting line 9 in Figure 18. The resulting distributions looked exactly like Figure 25.RHS. Further, only 64 solutions were found. Note that this observation is consistent with funnel theory: resolving three of the top treatments proposed by TAR2 constrained our system to one fifth of one percent of its original 35,228 behaviors.

4.3. Case Study C: Satellite Design

Our third example is much larger that then two previous. For reasons of confidentiality, the full details of this third model cannot be presented here. Further, this model uses so much domain-specific knowledge of satellite design that the general reader might learn little from its full exposition.

Analysts at the Jet Propulsion Laboratory sometimes debate design issues by building a semantic network connecting design decisions to requirements [14]. This network links *faults* and risk mitigation *actions* that effect a tree of *requirements* written by the stakeholders (see Figure 26). Potential faults within a project are modelled as influences on the edges between requirements. Potential fixes are modelled as influences on the edges between faults and requirements edges.



- Faces denote requirements;
- Toolboxes denote actions;
- Skulls denote faults;
- Conjunctions are marked with one arc; e.g. *require1* if *require2* and *require3*.
- Disjunctions are marked with two arcs; e.g. *fault1* if *fault2* or *fault3*.
- Numbers denote impacts; e.g. *action5* reduces the contribution of *fault3* to *fault1*, *fault1* reduces the impact of *require5*, and *action1* reduces the negative impact of *fault1*.
- Oval denotes structures that are expressible in the latest version of the JPL semantic net editor (under construction).

Figure 26. Above: a semantic net of the type used at JPL [18] Below: explanation of symbols.

This kind of requirements analysis seeks to maximize benefits (i.e., our coverage of the requirements) while minimizing the costs of the risk mitigation actions. Optimizing in this manner is complicated by the interactions inside the model - a requirement may be impacted by multiple faults, a fault may impact multiple requirements, an action may mitigate multiple faults, and a fault may be mitigated by multiple actions. For example, in Figure 26, *fault2* and *require4* are interconnected: if we cover *require4* then that makes *fault2* more likely which, in turn, makes *fault1* more likely which reduces the contribution of *require5* to *require3*.

The net can be executed by selecting actions and seeing what benefits results. One such network included 99 possible actions; i.e. $2^{99} \approx 10^{30}$ combinations of actions. The data cloud of Figure 1.ii was generated after 10,000 runs where each run selected at random from the 99 options. Note the wide variance in the possible behaviors.

		Cost				
	Benefit	400K	600K	800K	1,000K	Totals
	250		6	15	5	26
round 0	200	1	22	27	4	54
	150	1	6	5	1	13
	100		3	3		6
	50		1%			1
	Totals	2	38	50	10	100

		Cost				
	Benefit	400K	600K	800K	1,000K	Totals
	250	7	45	13		65
round 1	200	12	22	1		35
	150					
	100					
	50					
	Totals	19	67	14		100

		Cost				
	Benefit	400K	600K	800K	1,000K	Totals
	250	9	8	7		24
round 2	200	18	58			76
	150					
	100					
	50					
	Totals	27	66	7		101

		Cost				
	Benefit	400K	600K	800K	1,000K	Totals
	250	9	70	11		90
round 3	200	3	7			10
	150					
	100					
	50					
	Totals	12	77	11		100

		Cost				
	Benefit	400K	600K	800K	1,000K	Totals
	250	1	81	17		99
round 4	200		1			1
	150					
	100					
	50					
	Totals	1	82	17		100

Figure 27. Percentile matrices showing four rounds of incremental treatment learning for JPL satellite design. The data clouds for round 0 and round 4 appear as Figure 1.ii and Figure 1.iv (respectively).

The results of incremental treatment learning is shown in Figure 27. The first percentile matrix (called **round 0**) summarizes Figure 1.ii. As with all our other examples, as incremental treatment learning is applied, the variance is reduced and the mean values improve (compare **round 0** with **round 4** in Figure 27). In a result consistent with funnel theory, TAR2 could search a space of 1030 decisions to find 30 (out of 99) that crucially affected the cost/benefit of the satellite; i.e. TAR2 found 99-30=69 decisions that can be ignored [19].

For comparison purposes, a genetic algorithm (GA) was also applied to the same problem of optimized satellite design [48]. The GA also found decisions that generated high benefit, low cost projects. However, each such GA solution commented on every possible decision and there was no apparent way to ascertain which of these were the most critical decisions. The TAR2 solution was deemed superior to the GA solution by the domain experts, since the TAR2 solution required just 30 actions.

5. WHEN NOT TO USE INCREMENTAL TREATMENT LEARNING

Our approach is an inexpensive method of generating coarse-grained controllers from rapidly written models containing uncertainties. This kind of solution is inappropriate for certain classes of software such as mission critical or safety critical systems. For those systems, analysts should move beyond TAR2 and apply more elaborate modelling methods and extensive data collection to ensure exact and optimal solutions.

There are several other situations where incremental treatment learning should not be used. When trusted and powerful heuristics are available for a model, then a heuristic search for model properties may yield insight than random trashing within a model. Such heuristics might be modelled via (e.g.) fuzzy membership functions or Bayesian priors reflecting expert intuitions on how variables effect each other. Of course, such heuristics must be collected, assessed, and implemented. When the cost of such collection and assessment and implementation is too great, then our approach could be a viable alternative.

Also, our approach requires running models many thousands of times and therefore can't be applied to models that are too expensive or too slow to execute many times. For example:

- It may be too expensive or dangerous to conduct Monte Carlo simulations of in-situ process control systems for large chemical plants or nuclear power stations.
- Suppose some embedded piece of software must be run on a specialized hardware platform. In the case where several teams must access this

platform (e.g. the test team, the development team, the government certification team, and the deployment team), then it may be impossible to generate sufficient runs for incremental treatment learning.

- Many applications connect user actions on some graphical user interface to database queries and updates. Monte Carlo simulations of such applications may be very slow since each variable reference might require a slow disk access or a user clicking on some OK button. An ideal application suitable for incremental treatment learning comprises a *separate model* for the business logic which can be executed without requiring (e.g.) screen updates or database accesses.

6. RELATED WORK

6.1. Prior TAR2 Results

Other publications on treatment learning have assumed a one-shot use of TAR2 [24, 34-36]. This paper assumes an iterative approach. Our experience with users is that this iterative approach encourages their participation in the process and increases their sense of ownership in the conclusions.

6.2. Entropy-Based Learners

TAR2's treatments might be viewed as the attributes that most inform the decision making process. Holders of that view might therefore argue that treatments could be better formed using entropy measures of information content. Many machine learners have used such measures including the top-down decision tree induction algorithm C4.5 [45]. The attribute that offers the largest *information gain* is selected as the root of a decision tree. The example set is then divided up according to which examples do/do not satisfy the test in the root. For each divided example set, the process is then repeated recursively. The *information gain* of each attribute is calculated as follows. A tree C contain p examples of some class and n examples of other classes. The *information required* for the tree C is as follows [44]:

```
I(p,n)= -(p/(p+n))*$\log_2$(p/(p+n))
        -(n/(p+n))*$\log_2$(n/(p+n))
```

Say that some attribute A has values A1,A_2, . . . Av. If we select Ai as the root of a new sub-tree within C, this will add a sub-tree Ci containing those objects in C that have Ai. We can then define the expected value of the information required for that tree as the weighted average:

```
E(A)= (for i∈ values do
          E(A):= (E(A) + (($p_i$+$n_i$)/(p+n)*I($p_i$,$n_i$)))
lstat <= 11.66
|   rm <= 6.54
|   |   lstat <= 7.56 THEN  medhigh
|   |   lstat > 7.56
|   |   |   dis <= 3.9454
|   |   |   |   ptratio <= 17.6 THEN  medhigh
|   |   |   |   ptratio > 17.6
|   |   |   |   |   age <= 67.6 THEN  medhigh
|   |   |   |   |   age > 67.6 THEN  medlow
|   |   |   dis > 3.9454 THEN  medlow
|   rm > 6.54
|   |   rm <= 7.061
|   |   |   lstat <= 5.39 THEN  high}
|   |   |   lstat > 5.39
|   |   |   |   nox <= 0.435 THEN  medhigh}
|   |   |   |   nox > 0.435
|   |   |   |   |   ptratio <= 18.4 THEN  high
|   |   |   |   |   ptratio > 18.4 THEN  medhigh
|   |   rm > 7.061 THEN  high
lstat > 11.66
|   lstat <= 16.21
|   |   b <= 378.95
|   |   |   lstat <= 14.27 THEN  medlow
|   |   |   lstat > 14.27 THEN  low
|   |   b > 378.95 THEN  medlow
|   lstat > 16.21
|   |   nox <= 0.585
|   |   |   ptratio <= 20.9
|   |   |   |   b <= 392.92 THEN low
|   |   |   |   b > 392.92 THEN  medlow
|   |   |   ptratio > 20.9 THEN  low
|   |   nox > 0.585 THEN  low
```

Figure 28.A. A learnt decision tree from C4.5 from the 506 cases in HOUSING example from the UC Irvine machine learning repositor http://www.ics.uci.edu/~mlearn/MLRepository.html. The Classes for the houses are high, medhigh, medlow, and low. These classes indicate median value of owner-occupied homes in $1000's.

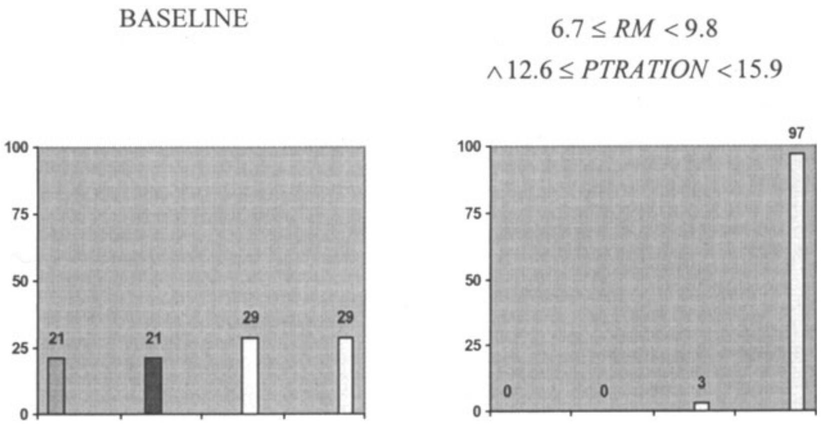


Figure 28.B. TAR2's output from the data used in Fig 28.A. The table shows the distributions of the classes in the example set (left-hand-side) and the results of the learnt treatment (right-hand-side). In the treatment PTRATIO denotes the pupil-teacher ratio by town and RM denotes the average number of rooms per dwelling.

The information gain of branching on A is therefore:

```
gain(A) = I(p, n) - E(A)
```

Figure 28.A shows the kind of decision tree generated using C4.5 from 506 examples. Figure 28.B shows the treatment learnt by TAR2 from the same data. Note that the treatment is much smaller that the tree learnt by C4.5. It turns out that C4.5's information measure is not the best method for forming treatments. Equation 3 selects attributes that most reduce the diversity of classes seen in the examples that fall into a subtree. Treatment learners needs a different kind of measure; i.e. one that that finds attribute ranges which occur more frequently in desired classes than in the undesired classes.

Decision tree learners like C4.5 can be used as a preprocessor to treatment learning. The TAR1 system (called TARZAN) [38] swung through the decision trees generated by C4.5 and 10-way cross-validation. TARZAN returned the smallest treatments that occurred in most of the ensemble that *increased* the percentage of branches leading to some preferred highly weighted classes and *decreased* the percentage of branches leading to lower weighted class. TAR2 was as experiment with applying TARZAN's tree pruning strategies directly to the C4.5 example sets. The resulting system is simpler, fast to execute, generates smaller theories that C4.5, and does not require calling a learner such as C4.5 as a subroutine.

6.3. Association Rule Learning

Another way to categorize TAR2 is a *weighted-class minimal contrast-set* association rule learner that uses *confidence measures* but not *support-based pruning*. This section discusses those terms.

Top-down decision tree classifiers like C4.5 and CART [7] learn rules with a single attribute pair on the right-hand side; e.g. *class= goodHouse*. Association rule learners like APRIORI [2] generate rules containing multiple attribute pairs on both the left-hand-side and the right-hand-side of the rules.

That is, classifiers have a small number of pre-defined targets (the classes) while, for association rule learners, the target is less constrained.

General association rule learners like APRIORI input a set of D transactions of items I and return associations between items of the form $LHS \Rightarrow RHS$ where $LHS \subset I$ and $RHS \subset I$ and $LHS \cap RHS = \emptyset$ In the terminology of APRIORI, an association rule has *support* s if s% of the D contains $X \wedge Y$; i.e. $s = |X \wedge Y| \ / \ |D|$ (where $|X \wedge Y|$ denotes the number of examples containing both X and Y). The *confidence* c of an association rule is the percent of transactions containing X which also contain Y ; i.e. $c=|X \wedge Y|/ \ |X|$. Many association rule learners use *support-based pruning* i.e. when searching for rules with high confidence, sets of items $I_i,\ldots,I_k$ are only be examined only if all its subsets are above some minimum support value.

Specialized association rule learners like CBA [28] and TAR2 impose restrictions on the right-hand-side. For example, TAR2's right-hand-sides show a prediction of the *change* in the class distribution if the constraint in the left-hand-side were applied. The CBA learner finds *class association rules*; i.e. association rules where the conclusion is restricted to one classification class attribute. That is, CBA acts like a classifier, but can process larger datasets that (e.g.) C4.5. TAR2 restricts the right-hand-side attributes to just those containing criteria assessment.

A common restriction with classifiers is that they assume the entire example set can fit into RAM. Learners like APRIORI are designed for data sets that need not reside in main memory. For example, Agrawal and Srikant report experiments with association rule learning using very large data sets with 10,000,000 examples and size 843MB [2]. However, just like Webb [53], TAR2 makes the memory-is-cheap assumption; i.e. TAR2 loads all it's examples into RAM.

Standard classifier algorithms such as C4.5 or CART have no concept of class *weighting*. That is, these systems have no notion of a *good* or *bad* class. Such learners therefore can't filter their learnt theories to emphasize the location of the *good* classes or *bad* classes. Association rule learners such as

MINWAL [9], TARZAN [38] and TAR2 explore *weighted learning* in which some items are given a higher priority weighting than others. Such weights can focus the learning onto issues that are of particular interest to some audience.

Support-based pruning is impossible in weighted association rule learning since with weighted items, it is not always true that subsets of *interesting* items (i.e. where the weights are high) are also interesting [9]. Another reason to reject support-based pruning is that it can force the learner to only miss features that apply to a small, but interesting subset of the examples [52]. Without support-based pruning, association rule learners rely on confidence-based pruning to reject all rules that fall below a minimal threshold of adequate confidence. TAR2's analogue of confidence-based pruning is the Δ measure shown in §3.

One interesting specialization of association rule learning is *contrast set learning*. Instead of finding rules that describe the current situation, association rule learners like STUCCO [4] finds rules that differ meaningfully in their distribution across groups. For example, in STUCCO, an analyst could ask, "What are the differences between people with Ph.D. and bachelor degrees?" TAR2's variant on the STUCCO strategy is to combine contrast sets with weighted classes with minimality. That is, TAR2 treatments can be viewed as the smallest possible contrast sets that distinguish situations with numerous highly-weighted classes from situations that contain more lowly-weighted classes.

6.4. Funnel Theory

Our development on funnel theory owes much to the deKleer's ATMS (assumption-based truth maintenance system) [16]. As new inferences are made, the ATMS updates its network of dependencies and sorts out the current conclusions into maximally consistent subsets (which we would call worlds). *Narrow funnels* are analogous to *minimal environments of small cardinality* from the ATMS research. However, funnels differ from the ATMS. Our view of funnels assumes a set-covering semantics and not the consistency-based semantics of the ATMS (the difference between these two views is detailed in [13]). The worlds explored by funnels only contain the variables seen in the subset of a model exercised by the supplied inputs. An ATMS world contains a truth assignment to every variable in the system. Consequently, the user of an ATMS may be overwhelmed with an exponential number of possible worlds. In contrast, our heuristic exploration of possible worlds, which assumes narrow funnels, generates a more succinct output. Further, the ATMS is only defined for models that generate logical justifications for each conclusion. Iterative treatment learning is silent on the form of the model: all it is concerned with is that a model, in whatever form, generates outputs that can be classified into desired and undesired behavior.

6.5. Fault Trees

We are not the first to note variability in knowledge extracted from users. Leveson [22] reports very large variances in the calculation of root node likelihood in software fault tree analysis:

- In one case study of 10 teams from 17 companies and 9 countries, the values computed for root node likelihood in trees from different teams differed by a factor of up to 36.
- When a unified fault tree was produced from all the teams, disagreements in the internal probabilities of the tree varied less, but still by a factor of 10.

The work presented here suggests a novel method to resolve Leveson's problem with widely varying root node likelihoods. If funnel theory is correct, then within the space of all disagreements in the unified fault tree, there exist a very small number of key values that crucially impact the root node likelihood. Using TAR2 the feuding teams could restrict their debates to just those key decisions.

6.6. Bayesian Reasoning

We do not use Bayesian reasoning for uncertain models for the same reason we don't use computational intelligence methods. Recall from our introduction that this work assumes *metrics starvation*: i.e. the absence of relevant domain expertise or specific numeric values in the domain being studied. Bayesian methods have been used to sketch out subjective knowledge (e.g. our software management oracle), then assess and tune that knowledge based on available data. Success with this method includes the COCOMO-II effort estimation tool [11] and defect prediction modelling [20]. In the domains where statistical data on cause-and-effect are lacking (e.g. our metrics starved domains), we have to approximate (i.e guess/make-up) some values to describe the model. Since there are too many uncertainties within the model, Bayesian reasoning may not yield stable result.

6.7. Simulation for Decision Making

Other research has explored simulation for making design decisions. Bricconi et al. [21] built a simulator of a server on a network, then assessed different network designs based on their impact on the server. Menzies and Sinsel assessed software project risk by running randomly selected combinations of inputs through a software risk assessment model [38].

Josephson et al. [26] executed all known options in a car design to find designs that were best for city driving conditions. Bratko et al. [6] built qualitative models of electrical circuits and human cardiac function. Where uncertainty existed in the model, or in the propagation rules across the model, a Bratko-style system would backtrack over all possibilities.

Simulation is usually paired with some summarization technique. Our research was prompted by certain short-comings with the summarization techniques of others. Josephson et al. used *dominance filtering* (a Pareto decision technique) to reduce their millions of designs down to a few thousand options. However, their techniques are silent on automatic methods for determining the difference between dominated and undominated designs. Bratko et al. used standard machine learners to summarize their simulations. Menzies and Sinsel attempted the same technique, but found the learnt theories to be too large to manually browse. Hence, they evolved a treequery language (TAR1) to find attribute ranges that were of very different frequencies on decision tree branches that lead to different classifications. TAR2 grew out of the realization that all the TAR1 search operations could be applied to the example set directly, without needing a decision tree learner as an intermediary. TAR2 is hence much faster than TAR1 (seconds, not hours).

7. CONCLUSION

When not all values within a model are known with certainty, analysts have at least three choices. Firstly, they can take the time to nail down those uncertain ranges. This is the preferred option. However, our experience strongly suggests that funding restrictions and pressing deadlines often force analysts to make decisions when many details are uncertain.

Secondly, analysts might use some sophisticated uncertainty reasoning scheme like Bayesian inference or the computational intelligence methods such as neural nets, genetic algorithms or fuzzy logic. These techniques require some minimal knowledge of expert opinion, plus perhaps some historical data to tune that knowledge. In situations of metrics starvation, that knowledge is unavailable.

This paper has explored a third option: try to understand a model by surveying the space of possible model behaviors.

Such a survey can generate a data cloud: a dense mass of possibilities with such a wide variance of output values that they can confuse, not clarify, the thinking of our analysts. However, in the case of data clouds generated from models containing narrow funnels, there exist key decisions which can condense that cloud.

Incremental treatment learning is a method for controlling the condensation of data clouds. At each iteration, users are presented with list of treatments that have most impact on a system. They select some of these and the results are added to a growing set of constraints for a model simulator. This human-in-the-loop approach increases user "buy-in" and allows for some human control of where a data cloud condenses. In the case studies shown above, data clouds where condensed in such a way as to decrease variance and improve the means of the behavior of the model being studied.

As stated in the introduction, there are several implications of this work. Even when we don't know exactly what is going on with a model, it is often possible to:

- Define minimal strategies that grossly decrease the uncertainty in that model's behavior.
- Identify which decisions are redundant; i.e. those not found within any funnel.

Also, when modelling is used to assist decision making, it is possible to reduce the cost of that modelling:

- Even hastily built models containing much uncertainty can be used for effective decision making.
- Further, for models with narrow funnels, elaborate and extensive and expensive data collection may not be required prior to decision making.

TAR2 exploits narrow funnels and is a very simple method for finding treatments at each step of iterative treatment learning. Iterative treatment learning is applicable to all models with narrow funnels. Empirically and theoretically, there is much evidence that many real-world models have narrow funnels. To test if a model has narrow funnels, it may suffice just to try TAR2 on model output. If a small number of key decisions can't be found, then iterative treatment should be rejected in favor of more elaborate techniques.

ACKNOWLEDGEMENTS

This paper benefited greatly from the remarks of the anonymous reviewers. This research was conducted at West Virginia University; the University of British Columbia, Vancouver, Canada; and at the Jet Propulsion Laboratory, California Institute of Technology under NASA contract NCC2-0979. The work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by

the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

REFERENCES

[1] A. Acree. On Mutations. PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1980.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In Proceedings of the 20th International Conference on Very Large Databases, 1994. Available from http://www.almaden.ibm.com/cs/people/ragrawal/papers/vldb94_rj.ps.

[3] T. Bäck, F. Hoffmeister, and H.-P. Schwefel. A survey of evolution strategies. In R. Belew and L. Booker, editors, Proceedings of the Fourth International Conference on Genetic Algorithms, pages 2-9, San Mateo, CA, 1991. Morgan Kaufman.

[4] S. Bay and M. Pazzani. Detecting change in categorical data: Mining contrast sets. In Proceedings of the Fifth InternationalConference on Knowledge Discovery and Data Mining, 1999. Available from http://www.ics.uci.edu/~pazzani/Publications/stucco.pdf.

[5] I. Bratko. Prolog Programming for Artificial Intelligence. (third edition). Addison-Wesley, 2001.

[6] I. Bratko, I. Mozetic, and N. Lavrac. KARDIO: a Study in Deep and Qualitative Knowledge for Expert Systems. MIT Press, 1989.

[7] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. Technical report, Wadsworth International, Monterey, CA, 1984.

[8] T. Budd. Mutation analysis of programs test data. PhD thesis, Yale University, 1980.

[9] C. Cai, A. Fu, C. Cheng, and W. Kwong. Mining association rules with weighted items. In Proceedings of International Database Engineering and Applications Symposium (IDEAS 98), August 1998. Available from http://www.cse.cuhk.edu.hk/òkdd/assoc_rule/paper.pdf.

[10] E. Chiang. Learning controllers for nonfunctional requirements, 2003. Masters thesis, University of British Columbia, Department of Electrical and Computer Engineering. In preparation.

[11] S. Chulani, B. Boehm, and B. Steece. Bayesian analysis of empirical software engineering cost models. IEEE Transaction on Software Engineerining, 25(4), July/August 1999.

[12] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, 2000.

[13] L. Console and P. Torasso. A Spectrum of Definitions of Model-Based Diagnosis. Computational Intelligence, 7:133-141, 3 1991.

[14] S. Cornford, M. Feather, and K. Hicks. Ddp a tool for lifecycle risk management. In IEEE Aerospace Conference, Big Sky, Montana, pages 441-451, March 2001.

[15] J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In AAAI '94, 1994.

[16] J. DeKleer. An Assumption-Based TMS. Artificial Intelligence, 28:163-196, 1986.

[17] C. Elkan. The paradoxical success of fuzzy logic. In R. Fikes and W. Lehnert, editors, Proceedings of the Eleventh National Conference on Artificial Intelligence, pages 698-703, Menlo Park, California, 1993. AAAI Press.

[18] M. Feather, H. In, J. Kiper, J. Kurtz, and T. Menzies. First contract: Better, earlier decisions for software projects. In ECE UBC tech report, 2001. Available from http://tim.menzies.com/pdf/01first.pdf.

[19] M. Feather and T. Menzies. Converging on the optimal attainment of requirements. In IEEE Joint Conference On Requirements Engineering ICRE'02 and RE'02, 9-13th September, University of Essen, Germany, 2002. Available from http://tim.menzies.com/pdf/02re02.pdf.

[20] N. E. Fenton and M. Neil. A critique of software defect prediction models. IEEE Transactions on Software Engineering, 25(5):675ñ689, 1999. Available from: http://citeseer.nj.nec.com/fenton99critique.html.

[21] E. T. G. Bricconi, E. Di Nitto. Issues in analyzing the behavior of event dispatching systems. In Proceedings of the 10th International Workshop on Software Specification and Design (IWSSD-10), San Diego, USA, November 2000.

[22] M. Heimdahl and N. Leveson. Completeness and consistency analysis of state-based requirements. IEEE Transactions on Software Engineering, May 1996.

[23] J. Horgan and A. Mathur. Software testing and reliability. In M. R. Lyu, editor, The Handbook of Software Reliability Engineering, pages 531-565, McGraw-Hill, 1996.

[24] Y. Hu. Better treatment learning, 2003. Masters Thesis, Department of Electrical Engineering, University of British Columbia, in preparation.

[25] J. Jahnke and A. Zundorf. Rewriting poor design patterns by good design patterns. In Proc. of ESEC:FSE '97 Workshop on Object-Oriented Reengineering, 1997.

[26] J. Josephson, B. Chandrasekaran, M. Carroll, N. Iyer, B. Wasacz, and G. Rizzoni. Exploration of large design spaces: an architecture and preliminary results. In AAAI '98, 1998. Available from: http://www.cis.ohio-state.edu/òjj/Explore.ps.

[27] A. Kakas, R. Kowalski, and F. Toni. The role of abduction in logic programming. In C. H. D.M. Gabbay and J. Robinson, editors, Handbook of Logic in Artificial Intelligence and Logic Programming 5, pages 235-324. Oxford University Press, 1998.

[28] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In KDD, pages 80ñ86, Sept 1998. Available from: http://citeseer.nj.nec.com/liu98integrating.html.

[29] R. Lutz and R. Woodhouse. Bi-directional analysis for certification of safety-critical software. In 1st International Software Assurance Certification Conference (ISACC'99), 1999. Available from http://www.cs.iastate.edu/òrlutz/publications/isacc99.ps.

[30] T. Menzies. Practical machine learning for software engineering and knowledge engineering. In Handbook of Software Engineering and Knowledge Engineering.World-Scientific, December 2001. Available from http://tim.menzies.com/pdf/00ml.pdf.

[31] T. Menzies and P. Compton. Applications of abduction: Hypothesis testing of neuroendocrinological qualitative compartmental models. Artificial Intelligence in Medicine, 10:145-175, 1997. Available from http://tim.menzies.com/pdf/96aim.pdf.

[32] T. Menzies, S. Easterbrook, B. Nuseibeh, and S. Waugh. An empirical investigation of multiple viewpoint reasoning in requirements engineering. In RE '99, 1999. Available from: http://tim.menzies.com/pdf/99re.pdf.

[33] T. Menzies and Y. Hu. Constraining discussions in requirements engineering. In First International Workshop on Model-basedRequirements Engineering, 2001. Available from: http://tim.menzies.com/pdf/01lesstalk.pdf.

[34] T. Menzies and Y. Hu. Reusing models for requirements engineering. In First International Workshop on Model-based RequirementsEngineering, 2001. Available from http://tim.menzies.com/pdf/01reusere.pdf.

[35] T. Menzies and Y. Hu. Agents in a wild world. In C. Rouff, editor, Formal Approaches to Agent-Based Systems, book chapter, 172002. Available from: http://tim.menzies.com/pdf/01agents.pdf.

[36] T. Menzies and Y. Hu. Just enough learning (of association rules). In WVU CSEE tech report, 2002. Available from http://tim.menzies.com/pdf/02tar2.pdf.

[37] T. Menzies and H. Singh. Many maybes mean (mostly) the same thing. In 2nd International Workshop on Soft Computing applied to Software Engineering (Netherlands), February, 2001. Available from http://tim.menzies.com/pdf/00maybe.pdf.

[38] T. Menzies and E. Sinsel. Practical large scale what-if queries: Case studies with software risk assessment. In Proceedings ASE 2000, 2000. Available from http://tim.menzies.com/pdf/00ase.pdf.

[39] T. Menzies and S. Waugh. On the practicality of viewpointbased requirements engineering. In Proceedings, Pacific Rim Conference on Artificial Intelligence, Singapore. Springer- Verlag, 1998. Available from http://tim.menzies.com/pdf/98pracai.pdf.

[40] C. Michael. On the uniformity of error propagation in software. In Proceedings of the 12th Annual Confererence on ComputerAssurance (COMPASS '97) Gaithersburg, MD, 1997.

[41] A. Parkes. Lifted search engines for satisfiability, 1999.

[42] D. Parnas. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 5(12):1053-1058, Dec. 1972.

[43] M. Paulk, B. Curtis, M. Chrissis, and C. Weber. Capability maturity model, version 1.1. IEEE Software, 10(4):18-27, July 1993.

[44] R. Quinlan. Induction of decision trees. Machine Learning, 1:81-106, 1986.

[45] R. Quinlan. C4.5: Programs for Machine Learning. Morgan Kaufman, 1992. ISBN: 1558602380.

[46] R. Rymon. An SE-tree based characterization of the induction problem. In International Conference on Machine Learning, pages 268-275, 1993.

[47] R. Rymon. An se-tree-based prime implicant generation algorithm. In Annals of Math. and A.I., special issue on Model-Based Diagnosis, volume 11, 1994. Available from http://citeseer.nj.nec.com/193704.html.

[48] J. D. . M. F. S. Cornford. Optimizing the design of end-to-end spacecraft systems using risk as a currency. In IEEE Aerospace Conference, Big Sky Montana, pages 9-16, March 2002.

[49] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

[50] H. T. Siegelmann and E. D. Sontag. On the computational power of neural nets. Journal of Computer and System Sciences, 50(1):132-150, 1995.

[51] J. Singer, I. P. Gent, and A. Smaill. Backbone fragility and the local search cost peak. Journal of Arti£cial Intelligence Research, 12:235-270, 2000.

[52] K. Wang, Y. He, D. Cheung, and F. Chin. Mining confident rules without support requirement. In 10th ACM International Conference on Informationand Knowledge Management(CIKM 2001), Atlanta, 2001. Available from http://www.cs.sfu.ca/òwangk/publications.html.

[53] G. Webb. Efficient search for association rules. In Proceeding of KDD-2000 Boston, MA, 2000. Available from http://citeseer.nj.nec.com/webb00efficient.html.

[54] W. Wong and A. Mathur. Reducing the cost of mutation testing: An empirical study. The Journal of Systems and Software, 31(3):185-196, December 1995.

[55] L. A. Zadeh. Outline of a new approach to the analysis of complex systems and decision processes. IEEE Transactions onSystems, Man and Cybernetics, SMC-3(1):28-44, Jan. 1973.